Summer 5-13-2019

Shared-Environment Call-by-Need

George W. Stelle
University of New Mexico

George Widgery Stelle

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Darko Stefanovic , Chairperson

Kei Davis

Stephanie Forrest

Stephen L. Olivier

Patrick Bridges

Shared-Environment Call-by-Need

by

George Widgery Stelle

B.S., University of British Columbia, 2008
M.S., Computer Science, University of New Mexico, 2013

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July, 2019

Dedication

For Beth

Acknowledgments

The list of people who deserve acknowledgment for this dissertation is far too long to enumerate here. Instead, I'll make a meager attempt at mentioning a few who played particularly important roles.

Starting at the beginning, I'd like to thank my parents and brothers, who gave me a childhood rich in love and fun. In a sense, my decision to pursue a Ph.D. was driven by a need to continue having fun doing what I love.

Most of the fun I've had has not been sitting and thinking about hard problems, but making lifelong friends along the way. To Taylor, Drew, Ben, Eric, George, Vu, and others, thank you for the technical discussions, the parties, the games, the hikes, the road trips, and most of all your friendship. Also, thanks to my three amigos for life, Drake, Mike, and Larkin, for the much-needed breaks from work.

I thank my Ph.D. advisor, Darko Stefanovic, for his support and advice through my unconventional final years through the program. Without his help and wisdom there is no question that this dissertation would not exist. Stephanie Forrest deserves a great deal of credit as well, both for bringing me into the department, and supporting and advising me selflessly through the challenging task of finding a topic I love. I would not be here in New Mexico at all without the help of my friend Eric Vatikiotis-Bateson.

The papers that form this basis of this dissertation would not have been possible without the help of my extraordinary co-authors, Darko Stefanovic, Stephen Olivier, and Stephanie Forrest [43, 42]. Their ability to write clearly and carefully continues to inspire me to be a better communicator. My committee members, which include the above co-authors, as well as Kei Davis and Patrick Bridges, have my thanks for their role in improving the dissertation.

I thank my wonderful wife, Beth, for being my partner in life and always inspiring me to be better. And to my children, Adelaide and Sullivan, thank you for making life so much more.

Shared-Environment Call-by-Need

by

George Widgery Stelle

B.S., University of British Columbia, 2008
M.S., Computer Science, University of New Mexico, 2013

Ph.D., Computer Science, University of New Mexico, 2019

Abstract

Call-by-need semantics formalize the wisdom that work should be done at most once. It frees programmers to focus more on the correctness of their code, and less on the operational details. Because of this property, programmers of lazy functional languages rely heavily on their compiler to both preserve correctness and generate high-performance code for high level abstractions. In this dissertation I present a novel technique for compiling call-by-need semantics by using shared environments to share results of computation. I show how the approach enables a compiler that generates high-performance code, while staying simple enough to lend itself to formal reasoning. The dissertation is divided into three main contributions. First, I present an abstract machine, the $\mathscr{CE}$ machine, which formalizes the approach. Second, I show that it can be implemented as a native code compiler with encouraging performance results. Finally, I present a verified compiler, implemented in the Coq proof assistant, demonstrating how the simplicity of the approach enables formal verification.

Contents

List of Figures

Chapter 1

Introduction

[L]azy functional programming is too important to be relegated to second-class citizenship. It is perhaps the most powerful glue functional programmers possess.

John Hughes

The strength of lazy functional programming languages is the freedom they give the programmer to focus on correctness instead of operational details. In a strict language, the programmer specifies what code will run, and when. In a lazy language, the programmer only specifies what the result should be, leaving the compiler responsible for ensuring that only code that is needed will be executed. Thanks to this freedom from operational concerns, there are two properties that lazy functional programmers tend to have. First, they reason about the correctness of their code to a degree seen almost nowhere else in the programming community [19, 41]. Second, they rely on compilers to generate efficient code in a way that programmers of strict languages don't. Essentially, they are leaving more operational decisions up to the compiler, and focusing their energy more on the correctness of their code. To paraphrase John Hughes, laziness is an essential tool for modular programming. As anyone who has done formal reasoning about programs knows, modular programming is essential for reasoning about programs.

It is for this reason that we compiler implementors must take great care in the design of our compilers for lazy languages. We must build compilers that generate efficient code: our programmers are relying particularly heavily on our ability to generate efficient code. We also must ensure that our compilers are correct: because lazy functional programmers are free to reason about the correctness of their code, we must ensure that any additional reasoning is not invalidated by bugs in the compiler. To illustrate this point: if a lazy functional programmer has time to prove ten theorems about his programs, while the strict language programmer only has time to prove three, then a bug in the lazy compiler may invalidate ten theorems, while a bug in the strict compiler may only invalidate three.

This dissertation presents a tool for attaining these two goals: a novel technique for implementing lazy semantics using shared environments, formalized as the $\mathscr{CE}$ machine. Essentially, the $\mathscr{CE}$ machine repurposes shared environments to share the results of computation. The thesis of this dissertation is that this approach helps to enable compilers that achieve these two goals. I explore the performance of the approach by implementing a native code compiler with encouraging results. This addresses the goal of high-performance code generation. To verify correctness, I take advantage of the simplicity of the approach to ease the proof burden, and implement a verified compiler using the Coq proof assistant. These two implementations provide evidence that the $\mathscr{CE}$ machine is a powerful tool for implementing lazy functional programming languages.

1.1 Outline

This dissertation is organized into six chapters. In this chapter, I provide an introduction to the dissertation, including an outline of the structure, instructions for access to artifacts and reproduction of results, a retrospective, and an overview of the contributions. Chapter 2 provides necessary background for understanding the dissertation, as well as further discussion of motivation. Chapter 3 defines and explains the $\mathscr{CE}$ machine, in both big and small-step semantics. Chapter 4 describes the implementation of a native code compiler based on the $\mathscr{CE}$ machine, and analyzes and discusses its performance. Chapter 5 presents a verified compiler, discussing the structure of the compiler and proofs. Finally, Chapter 6 discusses threats to validity, future work, and conclusions. The appendices are used to give further implementation details, both for the native code compiler (Appendix A) and the verified compiler (Appendix B). In the case of the native code compiler, the appendix shares some of the miscellaneous interesting properties of the implementation. For the verified compiler, the purpose of the appendix is to give the reader a fuller understanding of the structure and definitions involved in the proofs.

1.2 Reproducibility and Artifacts

The implementations presented in this dissertation are available for download to allow the reader to verify any claims made. All of the software is bundled as a single tarball at `http://cs.unm.edu/~stelleg/cem.tgz`. Instructions are included for building, running, and proof-checking the code. For performance results, the hardware and operating system are listed in Chapter 4. In addition to the above tarball, each implementation continues to be developed at `https://github.com/stelleg/cem` and `https://github.com/stelleg/cem_coq`. Finally, there is a simpler native code compiler for pedagogical purposes, available at `https://github.com/stelleg/cem_pearl`.

1.3 On Laziness

Because this work is focused on implementing call-by-need semantics, it is worth spending some time discussing why we care about lazy evaluation. The focus here is on high-level reasoning and opining, leaving a more technical coverage of the topic for Section 2.2, which defines and contrasts different evaluation strategies.

One easy argument for the importance of call-by-need is that it underlies the widely used programming language Haskell. Technically, Haskell is a non-strict language. This implies that both call-by-name and call-by-need are valid implementation strategies. In practice, there are some situations when one would prefer call-by-name: when storing an intermediate value is more expensive than re-computing it. This implies that in theory, Haskell could switch between call-by-name and call-by-value depending on the situation. In practice, implementations effectively always choose call-by-need, sometimes even performing compile-time transformations that increase sharing [33].

Even amongst the Haskell community, the advantages and disadvantages of lazy evaluation are hotly debated. For example, there exist both strictness annotations, and even strict-by-default variants of Haskell. There are real reasons for preferring strict evaluation in some contexts. In particular, reasoning about time and space requirements for lazy programs is notoriously difficult. As a result, there are cases when the time and space requirements can be surprisingly high.

The advantages of lazy semantics are most apparent when attempting to write high-level, composable abstractions. This is a strong argument for code re-use advantages in non-strict languages: by using laziness, one avoids work, non-termination, and work-buffering where possible without additional programmer effort [19].

There are also well-known cases where composing lazy programs can result in better asymptotics than strict composition. Consider the well-known example of finding the minimal value in a list.

```
take 1 . sort
```

With lazy semantics, this can result in an $O(n)$ time implementation, while the strict implementation of compose will always result in an $O(n \log n)$ implementation (assuming an $O(n \log n)$ sort). This kind of asymptotic improvement is a direct result of the efficiencies gained by avoiding eager work.

1.4 Retrospective

This section tells the story of how this dissertation came to be. The hope is to convey to the reader some context for the structure and approach that the dissertation takes.

Everything started with an appreciation of lazy evaluation and a desire to know how it works. Thus began investigation into how call-by-need semantics are currently implemented. Inspired by presentations of simple call-by-need approaches, such as the three instruction machine and the lazy Krivine machine, as well as sophisticated approaches such as the STG machine, I was afflicted with a nagging feeling that *there must be a simpler, lazier way to implement call-by-need.* After a lot of experimenting and thought, I finally discovered the approach presented here. While I was optimistic about the performance of a compiler, I was most excited by the *simplicity* of the approach. It was so easy to write a compiler! After a couple of failed attempts at writing papers with the primary objective being to excite the reader about the simplicity of the approach, I decided to instead focus on more concrete properties.

The first was performance: I hypothesized that the approach would lead to cases where I could beat the state of the art. This was confirmed by both a virtual machine and a native code compiler. It was also clear to me that trying to build a high-performance compiler to outperform GHC on real-world code was likely to fail, and I explicitly avoided making that a goal of the dissertation. Instead, I focused on showing that there were cases that outperform flat environments, leaving integrating shared and flat environments for future work.

Once I had shown that there were performance benefits to the approach, I still wanted to somehow use the simplicity of the approach for some concrete benefit. Around this time, I became aware of the field of certified programming. I realized I could use the simplicity of the approach to make formal reasoning easier, and build the first verified compiler for call-by-need. This was monumentally difficult. With very little training in formal reasoning, and no training in dependent types and machine-checked proofs, it took a long time working on my own to gain the skills to implement a verified compiler. Much of the effort was due to being too ambitious. It is a relatively straightforward thing to formalize and state theorems. Even when you are certain of the truth of those theorems, it is an entirely different beast proving them in a machine-checked logic. Every proof, definition, and theorem included in the paper and in the Coq code was built on tens of aborted versions. Building the verified compiler was the hardest thing I've ever done, by far.

Looking back, it would have been nice to have the two implementations be combined into one. While nice in some respects, this combination is a daunting task. Implementing a full native code compiler is a challenge in itself, but specifying, implementing, and verifying a native code compiler is a massive undertaking. CompCert, a verified compiler that compiles the lower level language C, took multiple PhDs worth of work to complete [28]. That said, it would likely have worked to verify and export into Haskell fragments of the native code compiler. For example, multiple times through the implementation process, the core language was extended. Making such core changes in the presence of proofs of correctness would make for a painful process, something that would have reduced the amount of time available for experimenting with the implementation. Overall, I am content with the approach of this dissertation: two separate compilers, with one focused on performance and extensibility and the other focused on correctness. I leave combining the two for future work, as I discuss further in Section 6.2.

1.5 Contributions

There are three primary contributions of this dissertation.

- A novel technique for implementing call-by-need semantics using shared environments, presented in Chapter 3. The technique is formalized as the $\mathscr{CE}$ machine, defined with both a big and small-step semantics.
- A full native code compiler from a simple lazy functional language with literals and primitive operations to x86_64 machine code, presented in Chapter 4. The implementation follows naturally from the definition of the $\mathscr{CE}$ machine. I show that the compiler performs comparably to the state of the art on a number of benchmarks. This implementation and its analysis provide evidence supporting the thesis that shared environment call-by-need has performance benefits in some cases over existing approaches.
- A verified compiler, presented in Chapter 5, that compiles call-by-need lambda calculus to a simple instruction machine, along with a specification of correctness and a proof that the compiler adheres to that specification. The compiler is implemented and the proofs checked in Coq, mechanizing the $\mathscr{CE}$ semantics in the process. This is the first verified compiler of a call-by-need semantics.

This implementation and mechanized proof provides evidence for the thesis that the simplicity of $\mathscr{CE}$ implementations lends itself to formal verification

Combined, these contributions support the core thesis of this dissertation: that shared environment call-by-need has valuable contributions to make to the study and implementation of call-by-need compilers. Smaller, more implementation-specific contributions are enumerated in Chapters 4 and 5.

Chapter 2

Background

Science is the belief in the
ignorance of experts.

Richard Feynman

This chapter provides relevant background for the $\mathscr{CE}$ machine and its two implementations, outlining lambda calculus, evaluation strategies, Curien's calculus of closures, and verifying implementations in formal logic.

2.1 Preliminaries

We begin with the simple lambda calculus [5]:

$$t ::= x \mid \lambda x.t \mid t\ t$$

where x is a variable, $\lambda x.t$ is an abstraction, and $t\ t$ is an application. We will primarily use lambda calculus with de Bruijn indices, which replaces variables with a natural number indexing into the binding lambdas. This calculus is given by the syntax:

$$t ::= i \mid \lambda t \mid t\ t$$

where $i \in \mathbb{N}$. In both cases, we use the standard Barendregt syntax conventions, namely that applications are left associative and the bodies of abstractions extend as far as possible to the right [5]. A *value* in lambda calculus refers to an abstraction. We are concerned only with evaluation to weak head normal form (WHNF), which terminates on an abstraction without entering its body.

In mechanical evaluation of expressions, it would be too inefficient to perform explicit substitution. To solve this, the standard approach uses closures [25, 11, 34, 7]. Closures combine a term with an environment, which binds the free variables of the term to closures. *Entering* a closure refers to the operational process of beginning to evaluate its term in its environment.

Because of its simplicity and its use of de Bruijn indices, we use Curien's calculus of closures [11] as the formal basis for closures, defined in Figure 2.1. It is a formalization of closures with an environment represented as a list of closures, indexed by de Bruijn indices. We will occasionally modify this calculus by replacing the de Bruijn indices with variables for readability, in which case variables are looked up in the environment instead of indexed, e.g., $t[x = c, y = c'])$ [5]. We also add superscript and subscript markers to denote unique syntax elements, e.g., $t', t_1 \in$ Term.

2.2 Evaluation Strategies

There are three standard evaluation strategies for lambda calculus: call-by-value, call-by-need, and call-by-name. Call-by-value evaluates every argument to a value, whereas call-by-need and call-by-name only evaluate an argument if it is needed. If an argument is needed more than once, call-by-name re-computes the value, whereas

Syntax

$$\begin{aligned} t, v &::= i \mid \lambda t \mid t\,t && \text{(Term)} \\ i &\in \mathbb{N} && \text{(Variable)} \\ c &::= t\,[\rho] && \text{(Closure)} \\ \rho &::= \bullet \mid c \cdot \rho && \text{(Environment)} \end{aligned}$$

Semantics

$$\frac{\begin{array}{c} t_1\,[\rho] \Downarrow \lambda t_2\,[\rho'] \\ t_2\,[t_3\,[\rho] \cdot \rho'] \Downarrow v \end{array}}{t_1 t_3\,[\rho] \Downarrow v}$$

$$\frac{c_i \Downarrow v}{i\,[c_0 \cdot c_1 \cdot \ldots \cdot c_i \cdot \rho] \Downarrow v}$$

$$\frac{}{\lambda t\,[\rho] \Downarrow \lambda t\,[\rho]}$$

Figure 2.1: Curien's calculus of closures

call-by-need memoizes the value, so it is computed at most once. Thus, call-by-need attempts to embody the best of both worlds—never repeat work (call-by-value), and never perform unnecessary work (call-by-name). These are intuitively good properties to have, illustrated by the following example, modified from Danvy et al. [13]:

$$\overbrace{c_m(c_m(\cdots(c_m}^{m}\ id\ \overbrace{id)\cdots)id)}^{m}\ \mathit{true}\ id\ \mathit{bottom}$$

where $c_n = \lambda s.\lambda z.\overbrace{s\,(s\cdots(s}^{n}\ z)\cdots)$, $\mathit{true} = \lambda t.\lambda f.t$, $id = \lambda x.x$, and $\mathit{bottom} = (\lambda x.x\,x)\lambda x.x\,x$. When evaluating this expression, call-by-value never terminates, call-by-name takes exponential time, and call-by-need takes only polynomial time [13]. Of course, this is a contrived example, but it illustrates desirable properties of call-by-need.

In practice, however, there are significant performance issues with call-by-need evaluation. We focus on the following: *delaying a computation and performing it later is slower than performing it immediately.* This deficiency is well understood [22, 34], and is part of the motivation for *strictness analysis* [30, 47], which transforms non-strict evaluation to strict when possible.

When compiling applications, there are two general implementation approaches. In the first, *eval/apply*, the caller first *evaluates* the function, then *applies* the arguments to it by knowing its arity. In the second, *push/enter*, the caller *pushes* the arguments onto the stack, then *enters* the code that will evaluate to a function [29].

2.3 Existing Call-by-Need Machines

Diehl et al. [14] review the call-by-need literature in detail. Here we summarize the most relevant points.

The most well known and widely used machine for lazy evaluation is the Spineless Tagless G-Machine (STG machine), which underlies the Glasgow Haskell Compiler (GHC). STG uses flat environments that can be allocated on the stack, the heap, or some combination [34].

Two other influential lazy evaluation machines relevant to the $\mathscr{CE}$ machine are the call-by-need Krivine machine [17, 23, 38], and the three-instruction machine (TIM) [16]. Krivine machines started as implementing call-by-name evaluation, and were later extended to call-by-need [23, 38, 13, 17]. The $\mathscr{CE}$ machine modifies the lazy Krivine machine to capture the environment sharing given by the cactus environment. The TIM is an implementation of call-by-need and call-by-name [16]. It involves, as the name suggests, three machine instructions, `TAKE`, `PUSH`, and `ENTER`. In Section 4.1, We follow Sestoft [38] and re-appropriate these instructions for the $\mathscr{CE}$ machine.

There has also been recent interest in *heapless* abstract machines for lazy evaluation. Danvy et al. [12] and Garcia et al. [37] independently derived similar machines from the call-by-need lambda calculus [4]. These are interesting approaches, but it is not yet clear how these machines could be implemented efficiently.

2.4 Formal Logic

With recent improvements in higher order logics, machine verification of algorithms has become a valuable tool in software development. Instead of relying heavily on tests to check the correctness of programs, verification can prove that algorithms implement their specification for *all* inputs. Implementing both the specification and the proof in a machine-checked logic removes the vast majority of bugs found in hand-written proofs, ensuring far higher confidence in correctness than other standard methods. Other approaches, such as fuzz testing, have confirmed empirically that verified programs remove all bugs [49].

This approach applies particularly well to compilers. Often, the specification for a compiler is complete: source level semantics for some languages are exceedingly straightforward to specify, and target architectures have lengthy specifications that are amenable to implementation in a machine-checked logic. In addition, writing tests for compilers that cover all cases is even more hopeless than most domains, due to the size and complexity of the domain and codomain. The return on investment is also high: all reasoning about programs compiled with a verified compiler is provably preserved.

Likely due to the complexities discussed above involved in implementing lazy languages, among other factors, existing work on verification has focused on compiling strict languages [10, 28, 24]. In this dissertation we use the simple $\mathscr{CE}$ machine as a base for a verified compiler of a lazy language, using the Coq proof assistant.

As with many areas of research, the devil is in the details. The details, in this context, are the specification of a what it means to be a verified compiler. Generally speaking, a verified compiler of a functional language is one that preserves computation of values. That is, we have an implication: *if the source semantics denotes a value, then the compiled code computes an equivalent value* [10]. The important thing to note is that the implication is only in one direction. If the source semantics never terminates, this class of correctness theorem says nothing about the behavior of the compiled code. This has consequences for Turing-complete source languages. If we are unsure if a source program terminates and wish to run it to check experimentally if it does, then if we run the compiled code and it returns a value, we cannot be certain that it corresponds to a value computed in the source semantics.

While in theory one could solve this by proving the implication in the other direction, that is, *if the compiled code computes a value then the source semantics computes an equivalent value*, in practice this is prohibitively difficult. Effectively, the induction rules for the abstract machine make constructing such a proof challenging, as we discuss further in Chapter 5.

One approach for getting around this issue is to try and capture the divergent behavior by defining a diverging semantics explicitly [32]. Then one can safely claim that *if the source semantics diverges according to our diverging semantics, then the compiled code also diverges.*

This dissertation takes the approach of Chlipala [10] and defines verification as the first implication above, focusing on the case in which the source semantics evaluates to a value. This is still a strong result: any source program that has meaning compiles to an executable with equivalent meaning. In addition, if anyone ever chooses to augment the language with a type system that ensures termination, or some notion of progress, then they could use that in combination with our verification proof.

2.5 Environment Representations

As mentioned in Section 2.1, environments bind free variables to closures. While any implementation of an environment performs the same function, there is significant flexibility in how environments can be represented. In this section we review this design space in the context of existing work, both for call by value and call-by-need.[1]

There are two common approaches to environment representation: *flat* environments and *shared* environments (also known as linked environments) [2, 39]. A flat environment is one in which each closure has its own record of the terms its free variables are bound to. A shared environment is one in which parts of that record can be shared among multiple closures [2, 39]. For example, consider the following term:

$$(\lambda x.(\lambda y.t_0)(\lambda z.t_1))t_2$$

Assuming the term t_0 has both x and y as free variables, we must evaluate it in an environment that binds both x and y. Similarly, assuming t_1 contains both z and x as free variables, we must evaluate it in an environment containing bindings for both x and z. Thus, we can represent the closures for evaluating t_0 and t_1 as

$$t_0[x = t_2[\bullet], y = c]$$

[1]Some work refers to this space as *closure* representation rather than *environment* representation [39, 2]. Because the term part of the closure is simply a code pointer and the interesting design choices are in the environment, we refer to the topic as environment representation. Note also that global variables are often omitted from environments in real implementations, we don't consider this implementation detail here.

and

$$t_1[x = t_2[\bullet], z = c_1]$$

respectively, where $\bullet$ is the empty environment. These are examples of *flat* environments, where each closure comes with its own record of all of its free variables. Because of the nested scope of the given term, x is bound to the same closure in the two environments. Thus, we can also create a shared, linked environment, represented by the following diagram:

$$\begin{tikzcd}
& \bullet & \\
& x = t_2[\bullet] \arrow[u] & \\
y = c \arrow[ur] & & z = c_1 \arrow[ul]
\end{tikzcd}$$

Now each of the environments is represented by a linked list, with the binding of x shared between them. This is an example of a *shared* environment [2]. This shared, linked structure dates back to the first machine for evaluating expressions, Landin's SECD machine [25].

The drawbacks and advantages of each approach are well known. With a flat environment, variable lookup can be performed with a simple offset [34, 1]. On the other hand, significant duplication can occur, as we discuss in Section 2.6. With a shared environment, that duplication is removed, but at the cost of possible link traversal upon dereference.

As with most topics in compilers and abstract machines, the design space is actually more complex. For example, Appel and Jim show a wide range of hybrids [2] between the two, and Appel and Shao [39] show an optimized hybrid that aims to achieve the benefits of both approaches. And as shown in the next section, choice of evaluation strategy further complicates the picture.

2.6 Existing Call-by-Need Environments

Existing call-by-need machines use flat environments with a heap of closures [34, 16, 22, 8]. These environments may contain some combination of primitive values and pointers into the heap (p below). The pointers and heap implement the memoization of results required for call-by-need. Returning to the earlier example, $(\lambda x.(\lambda y.t_0)(\lambda z.t_1))t_2$, we can view a simplified execution state for this approach when entering t_0 as follows:

Closure

$$t_0[x = p_0, y = p_1]$$

Heap

$$p_0 \mapsto t_2[\bullet]$$
$$p_1 \mapsto \lambda z.t_1[x = p_0]$$

Consider $t_2[\bullet]$, the closure at p_0. If it is not in WHNF (this sort of unevaluated closure is called a *thunk* [21, 35]), then if it is entered in either the evaluation of t_0 or t_1, the resulting value will overwrite the closure at p_0. The result of the computation is then shared with all other instances of x in t_0 and t_1. In the case that terms have a large number of shared variables, environment duplication can be expensive. Compile-time transformation [35] (tupling arguments) helps, but we show that the machine can avoid duplication completely.

Depending on t_0, either or both of the closures created for its free variables may not be evaluated. Therefore, it is possible that the work of creating the environment for that thunk will be wasted. This waste is well known, and existing approaches address it by avoiding thunks as much as possible [34, 22]. Unfortunately, in cases like the above example, thunks are necessary. We aim to minimize the cost of creating such thunks.

Thunks are special in another way. Recall that one advantage of flat environments is quick variable lookups. In a lazy language, this advantage is reduced because *a thunk can only be entered once.* After it is entered, it is overwritten with a value, so the next time that heap location is entered it is entered with a value and a different environment. Thus, the work to ensure that the variable lookup is fast is used for at most one evaluation of the thunk. This is in contrast to a call-by-value language, in which every closure is a value, and can therefore be entered an arbitrary number of times.

A more subtle drawback of the flat environment representation is that environments can vary in size, and thus a value in WHNF can be too large to fit in the space allocated for the thunk it is replacing. This problem is discussed in Jones et al. [34], where the proposed solution is to put the value closure in a fresh location in the heap where there is sufficient room. The original thunk location is then replaced with an indirection to the value at the freshly allocated location. These indirections are removed during garbage collection, but do impose some cost, both in runtime efficiency and implementation complexity.

We have thus far ignored a number of details with regard to current implementations. For example, the STG machine can split the flat environment, so that part is allocated on the stack and part on the heap. The TIM allocates its flat environments separately from its closures so that each closure is a code pointer, environment pointer pair [16] while the STG machine keeps environment and code co-located. Still, the basic design principle holds: a flat environment for each closure allows quick variable indexing, but with an initial overhead.

	Flat Environment	Shared Environment
Call-by-need	STG [34], TIM [16], GRIN [8]	$\mathscr{CE}$ Machine
Call-by-value	ZAM [27], SML/NJ [3]	ZAM, SECD [25], SML/NJ

Figure 2.2: Evaluation strategy and environment structure design space. Each acronym refers to an existing implementation. Some implementations use multiple environment representations.

To summarize, the flat environment representation in a call-by-need language implies that whenever a term might be needed, the necessary environment is constructed from the current environment. This operation can be expensive, and it is wasted if the variable is never dereferenced. In this work, we aim to minimize this potentially unnecessary overhead.

Figure 2.2 depicts the design space relevant to this chapter. There are existing call by value machines with both flat and shared environments, and call-by-need machines with flat environments. This is the first work to use a shared environment to implement lazy evaluation.

It is worth noting that there has been work on lazy machines that effectively use linked environments, which could potentially be implemented as a shared environment, e.g., Sestoft's work on Krivine machines [38], but none make the realization that the shared environment can be used to implement sharing of results, which is the primary contribution of this chapter.

Chapter 3

$\mathscr{CE}$ Machine

The lurking suspicion that something could be simplified is the world's richest source of rewarding challenges.

Edsger Dijkstra

This chapter defines the $\mathscr{CE}$ machine semantics, both big-step and small-step versions. It attempts to convey some intuition for why the shared environment structure works as a technique for sharing results of computation. The definitions here are the core of both implementations, the native code compiler in Chapter 4 and the verified compiler in Chapter 5. We formalize the connection between call-by-need evaluation and shared environments in a big-step semantics in Section 3.1. Section 3.2 implements the big-step with a small-step semantics by adding a context (or stack). The proof that it is a correct implementation is left for Chapter 5.

3.1 Big-Step $\mathscr{CE}$

This section shows how the shared environment approach can be applied to call-by-need evaluation. We start with a big step semantics that abstracts away environment representation, Curien's calculus of closures, and then shows how it can be modified to force sharing. Recall Curien's call-by-name calculus of closures in Figure 2.1. [1] The App rule pushes a closure onto the environment, and the Id rule indexes into the environment, entering the corresponding closure. We show that by removing ambiguity about how the environments are represented, and forcing them to be represented in a *parent pointer tree*, we can define a novel approach to call-by-need.

To start, consider again the example from Section 2.5, this time with de Bruijn indices: $(\lambda(\lambda t_0)\ (\lambda t_1))t_2$. The terms t_0 and t_1, when evaluated in Curien's calculus of closures, would have the following environments, respectively:

$$\begin{array}{c} c_0 \cdot t_2[\bullet] \cdot \bullet \\ c_1 \cdot t_2[\bullet] \cdot \bullet \end{array}$$

As with the named case, the second closure is identical in each environment. And again, we can represent these environments with a shared environment, this time keeping call-by-need evaluation in mind:

$$\begin{tikzcd}
& \bullet & \\
& t_2[\bullet] \arrow[u] & \\
c_0 \arrow[ur] & & c_1 \arrow[ul]
\end{tikzcd}$$

This inverted tree structure seen earlier with the leaves pointing toward the root is called a *cactus stack* (sometimes called a spaghetti stack or saguaro stack) when used

[1]Curien calls it a "lazy" evaluator, and there is some ambiguity with the term lazy, but here the term is used only to mean call-by-need. Curien's condition checking that $i < m$ is omitted as the semantics is only defined for closed terms.

to implement stacks [18, 20], or a parent pointer tree in general. In this use case, every node defines an environment as the sequence of closures in the path to the root. If $t_2[\bullet]$ is a thunk, and is updated in place with the value after its first reference, then both environments would contain the resulting value. This is exactly the kind of sharing that is required by call-by-need, and thus we can use this structure to build a call-by-need evaluator. This is the essence of the $\mathscr{CE}$ machine.

Curien's calculus of closures does not differentiate between flat and shared environment representations; it has no need to. Therefore, we must derive a new semantics, forcing the environment to be shared. Because we can hold the closure directly in the environment, the standard approach of a heap of closures is replaced with a *heap of environments*. To enforce sharing, we extend Curien's calculus of closures to explicitly include the heap of environments, which we refer to as a *cactus environment* ($\mathscr{CE}$). This cactus environment structure is a parent pointer tree of closures.

See Figure 3.2 for the syntax and semantics of the $\mathscr{CE}$ big step semantics. Recall that we are only concerned with evaluation of closed terms. The initial closed term t is placed in a $(t[0], \varepsilon[0 \mapsto \bullet])$ configuration, and evaluation terminates on a value. Some shorthand is used to make heap notation more palatable for both the big-step semantics presented here and the small step semantics presented in the next section. $\mu(l,i) = l' \mapsto c \cdot l''$ denotes that looking up the i'th element in the linked environment structure starting at l results in location l', where closure c and continuing environment l'' reside. $\mu(l) = c \cdot l'$ is the statement that $l \mapsto c \cdot l' \in \mu$, and $\mu(u \mapsto c \cdot l')$ is μ with location u updated to map to $c \cdot e$. Two different semantics are defined, one for call-by-name (Figure 3.1) and one for call-by-need (Figure 3.2). Having both makes the connection to Curien's call-by-name calculus more straightfoward. The rule for application is identical for both semantics: each evaluates the left hand side to a function, then binds the variable in the cactus environment, extending the current

Syntax

$$
\begin{aligned}
t &::= i \mid \lambda t \mid t\, t && \text{(Term)}\\
i &\in \mathbb{N} && \text{(Variable)}\\
c &::= t\,[l] && \text{(Closure)}\\
v &::= \lambda t\,[l] && \text{(Value)}\\
\mu &::= \varepsilon \mid \mu\,[l \mapsto \rho] && \text{(Heap)}\\
\rho &::= \bullet \mid c \cdot l && \text{(Environment)}\\
l, f &\in \mathbb{N} && \text{(Location)}\\
s &::= (c, \mu) && \text{(Configuration)}
\end{aligned}
$$

Semantics

$$\frac{\mu\,(l,i) = l' \mapsto c \cdot l'' \quad (c,\mu) \Downarrow (v,\mu')}{(i\,[l],\mu) \Downarrow (v,\mu')} \quad \text{(Id)}$$

$$\frac{\begin{array}{c}(t_0\,[l],\mu) \Downarrow (\lambda t_2\,[l'],\mu') \quad f \notin \operatorname{dom}(\mu') \\ (t_2\,[f],\mu'\,[f \mapsto t_3\,[l] \cdot l']) \Downarrow (v,\mu'')\end{array}}{(t_0\,t_3\,[l],\mu) \Downarrow (v,\mu'')} \quad \text{(App)}$$

$$\frac{}{(\lambda t\,[l],\mu) \Downarrow (\lambda t\,[l],\mu)} \quad \text{(Abs)}$$

Figure 3.1: Big-step call-by-name $\mathscr{CE}$ syntax and semantics

environment.

The only difference between this semantics and Curien's is that if we need to extend an environment multiple times, the semantics *requires* sharing it among the extensions. This makes no difference for call-by-name, but it is needed for the sharing of results in the Id rule. The explicit environment sharing ensures that the closure that is overwritten with a value is shared correctly.

Syntax

$$
\begin{aligned}
t &::= i \mid \lambda t \mid t\ t && \text{(Term)}\\
i &\in \mathbb{N} && \text{(Variable)}\\
c &::= t\,[l] && \text{(Closure)}\\
v &::= \lambda t\,[l] && \text{(Value)}\\
\mu &::= \varepsilon \mid \mu\,[l \mapsto \rho] && \text{(Heap)}\\
\rho &::= \bullet \mid c \cdot l && \text{(Environment)}\\
l, f &\in \mathbb{N} && \text{(Location)}\\
s &::= (c, \mu) && \text{(Configuration)}
\end{aligned}
$$

Semantics

$$\frac{\mu\,(l,i) = l' \mapsto c \cdot l'' \quad (c,\mu) \Downarrow (v,\mu')}{(i\,[l],\mu) \Downarrow (v, \mu'\,[l' \mapsto v \cdot l''])} \quad \text{(Id)}$$

$$\frac{\begin{array}{c}(t_0\,[l],\mu) \Downarrow (\lambda t_2\,[l'],\mu') \quad f \notin \mathrm{dom}\,(\mu')\\ (t_2\,[f],\mu'\,[f \mapsto t_3\,[l] \cdot l']) \Downarrow (v,\mu'')\end{array}}{(t_0\,t_3\,[l],\mu) \Downarrow (v,\mu'')} \quad \text{(App)}$$

$$\frac{}{(\lambda t\,[l],\mu) \Downarrow (\lambda t\,[l],\mu)} \quad \text{(Abs)}$$

Figure 3.2: Big-step call-by-need $\mathscr{CE}$ syntax and semantics

3.2 Small-Step $\mathscr{CE}$

Using the big-step $\mathscr{CE}$ from the previous section, we construct a small-step semantics by adding a stack. The syntax and semantics are defined in Figure 3.3.

The small-step semantics operate identically to the big-step, extended only with a context (or stack) to implement the updates from the Id subderivation ($\sigma\ u$) and the operands from the App subderivation ($\sigma\ c$). Much like the big-step semantics, a term t is inserted into an initial state $\langle t[0], \sigma, \varepsilon[0 \mapsto \bullet]\rangle$. For the update rule, the

Syntax

$$\begin{aligned}
s &::= \langle c, \sigma, \mu \rangle && \text{(State)}\\
t &::= i \mid \lambda t \mid t\, t && \text{(Term)}\\
i &\in \mathbb{N} && \text{(Variable)}\\
c &::= t\,[l] && \text{(Closure)}\\
v &::= \lambda t\,[l] && \text{(Value)}\\
\mu &::= \varepsilon \mid \mu\,[l \mapsto \rho] && \text{(Heap)}\\
\rho &::= \bullet \mid c \cdot l && \text{(Environment)}\\
\sigma &::= \square \mid \sigma\, c \mid \sigma\, u && \text{(Stack)}\\
l, u, f &\in \mathbb{N} && \text{(Location)}
\end{aligned}$$

Semantics

$$\begin{aligned}
\langle v, \sigma\, u, \mu \rangle &\to \langle v, \sigma, \mu\,(u \mapsto v \cdot l) \rangle \text{ where } c \cdot l = \mu\,(u) && \text{(Upd)}\\
\langle \lambda t\,[l], \sigma\, c, \mu \rangle &\to \langle t\,[f], \sigma, \mu\,[f \mapsto c \cdot l] \rangle f \notin \text{dom}\,(\mu) && \text{(Lam)}\\
\langle t\, t'\,[l], \sigma, \mu \rangle &\to \langle t\,[l], \sigma\, t'\,[l], \mu \rangle && \text{(App)}\\
\langle i\,[l], \sigma, \mu \rangle &\to \langle c, \sigma\, l'', \mu \rangle \text{ where } l'' \mapsto c \cdot l' = \mu\,(l, i) && \text{(Var)}
\end{aligned}$$

Figure 3.3: Small-step $\mathscr{CE}$ syntax and semantics

current closure is a value, and there is an update marker as the outermost context. This implies that a variable was entered and that the current closure represents the corresponding value for that variable. Thus, we update the location u that the variable entered, replacing whatever closure was entered with the current closure. The Lam rule takes an argument off the context and binds it to a variable, allocating a fresh heap location for the bound variable. This ensures that every instance of the variable will point to this location, and thus the bound closure will be evaluated at most once. The App rule simply pushes the argument term in the current environment. The Var rule enters the closure pointed to by the *i*'th environment location.

To get some intuition for the $\mathscr{CE}$ machine and how it works, please refer to Fig-

ure 3.4, which displays the steps in the evaluation of the term
$(\lambda a.(\lambda b.b\ a)\lambda c.c\ a)\ ((\lambda i.i)\lambda j.j)$, or $(\lambda(\lambda 0\ 1)\ \lambda 0\ 1)\ ((\lambda 0)\ \lambda 0)$ with de Bruijn indices.

$$
\begin{aligned}
& \langle(\lambda(\lambda 0\ 1)\ \lambda 0\ 1)\ ((\lambda 0)\ \lambda 0)[0], \Box, \varepsilon[0 \mapsto \bullet]\rangle \\
& \to \langle \lambda(\lambda 0\ 1)\ \lambda 0\ 1[0], \Box(\lambda 0)\ \lambda 0[0], \varepsilon[0 \mapsto \bullet]\rangle \\
& \to \langle (\lambda 0\ 1)\ \lambda 0\ 1[1], \Box, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0]\cdot 0]\rangle \\
& \to \langle \lambda 0\ 1[1], \Box\lambda 0\ 1[1], \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0]\cdot 0]\rangle \\
& \to \langle 0\ 1[2], \Box, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0]\cdot 0][2 \mapsto \lambda 0\ 1[1]\cdot 1]\rangle \\
& \to \langle 0[2], \Box 1[2], \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0]\cdot 0][2 \mapsto \lambda 0\ 1[1]\cdot 1]\rangle \\
& \to \langle \lambda 0\ 1[1], \Box 1[2]2, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0]\cdot 0][2 \mapsto \lambda 0\ 1[1]\cdot 1]\rangle \\
& \to \langle \lambda 0\ 1[1], \Box 1[2], \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0]\cdot 0][2 \mapsto \lambda 0\ 1[1]\cdot 1]\rangle \\
& \to \langle 0\ 1[3], \Box, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0]\cdot 0][2 \mapsto \lambda 0\ 1[1]\cdot 1][3 \mapsto 1[2]\cdot 1]\rangle \\
& \to \langle 0[3], \Box 1[3], \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0]\cdot 0][2 \mapsto \lambda 0\ 1[1]\cdot 1][3 \mapsto 1[2]\cdot 1]\rangle \\
& \to \langle 1[2], \Box 1[3]3, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0]\cdot 0][2 \mapsto \lambda 0\ 1[1]\cdot 1][3 \mapsto 1[2]\cdot 1]\rangle \\
& \to \langle 0[1], \Box 1[3]3, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0]\cdot 0][2 \mapsto \lambda 0\ 1[1]\cdot 1][3 \mapsto 1[2]\cdot 1]\rangle
\end{aligned}
$$

Figure 3.4: Small-step $\mathscr{CE}$ example. Evaluation of $(\lambda(\lambda 0\ 1)\ \lambda 0\ 1)\ ((\lambda 0)\ \lambda 0)$

$$\begin{aligned}
&\to \langle (\lambda 0)\ \lambda 0[0], \square 1[3]31, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1] \rangle \\
&\to \langle \lambda 0[0], \square 1[3]31\lambda 0[0], \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1] \rangle \\
&\to \langle 0[4], \square 1[3]31, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0] \rangle \\
&\to \langle \lambda 0[0], \square 1[3]314, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0] \rangle \\
&\to \langle \lambda 0[0], \square 1[3]31, \varepsilon[0 \mapsto \bullet][1 \mapsto (\lambda 0)\ \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0] \rangle \\
&\to \langle \lambda 0[0], \square 1[3]3, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto 1[2] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0] \rangle \\
&\to \langle \lambda 0[0], \square 1[3], \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0] \rangle \\
&\to \langle 0[5], \square, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0][5 \mapsto 1[3] \cdot 0] \rangle \\
&\to \langle 1[3], \square 5, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0][5 \mapsto 1[3] \cdot 0] \rangle \\
&\to \langle 0[1], \square 5, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0][5 \mapsto 1[3] \cdot 0] \rangle \\
&\to \langle \lambda 0[0], \square 51, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0][5 \mapsto 1[3] \cdot 0] \rangle \\
&\to \langle \lambda 0[0], \square 5, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0][5 \mapsto 1[3] \cdot 0] \rangle \\
&\to \langle \lambda 0[0], \square, \varepsilon[0 \mapsto \bullet][1 \mapsto \lambda 0[0] \cdot 0][2 \mapsto \lambda 0\ 1[1] \cdot 1][3 \mapsto \lambda 0[0] \cdot 1][4 \mapsto \lambda 0[0] \cdot 0][5 \mapsto \lambda 0[0] \cdot 0] \rangle
\end{aligned}$$

Figure 3.4: Small-step $\mathscr{CE}$ example. Evaluation of $(\lambda(\lambda 0\ 1)\ \lambda 0\ 1)\ ((\lambda 0)\ \lambda 0)$ (cont.)

Chapter 4

Native Code Compilation

Much of my work has come from being lazy.

John Backus

Existing implementations of call-by-need take care in *packaging* a delayed computation, or *thunk*, by building a closure with an array that contains the bindings of all free variables [34, 8]. The overhead induced by this operation is well known, and is one reason existing implementations avoid thunks wherever possible [22]. A key insight of the $\mathscr{CE}$ Machine is that this overhead can be minimized by only recording a location in a shared environment.

As an example, consider the application $f\ e$. In existing call-by-need implementations, e.g., the STG machine[34], a closure with a flat environment will be constructed for e. Doing so incurs a time and memory cost proportional to the number of free variables of e. [1] We minimize this packaging cost by recording a location in a shared environment, which requires only two machine words (and two instructions) for the

[1]In some implementations, these are lambda-lifted to be formal parameters, but the principle is the same.

thunk: one for the code pointer, and one for the environment pointer. One way to think about the approach is that it is *lazier* about lazy evaluation: in the case that e is unneeded, the work to package it in a thunk is entirely wasted. In the spirit of lazy evaluation, we attempt to minimize this potentially unnecessary work.

This chapter presents a simple implementation of the $\mathscr{CE}$ machine that compiles a simple lazy functional language to x86_64 assembly. In addition, it presents a preliminary evaluation that shows performance comparable to existing implementations (Sections 4.1 and 4.2).

Section 4.1 describes a straightforward implementation of $\mathscr{CE}$, extended with machine literals and primitive operations, and compiling directly to native code. Section 4.2 evaluates the implementation, showing that it is capable of performing comparably to existing implementations despite lacking several common optimizations, and discusses the results. Section 4.3 discusses related work, limitations of the approach, and some ideas for future work.

4.1 Implementation

This section describes how the $\mathscr{CE}$ machine can be mapped directly to x86_64 instructions. Specifically, we re-define the three instructions given by the TIM [16]: `TAKE`, `ENTER`, and `PUSH`, and implement them with x86_64 assembly. We also describe several design decisions, as well as some optimizations. All implementation and benchmark code is available at `http://cs.unm.edu/~stelleg/cem.tgz`.

Each closure is represented as a ⟨code pointer, environment pointer⟩ tuple. The Context is implemented as a stack, with updates represented as a ⟨null pointer, environment pointer⟩ tuple to differentiate them from closure arguments. The Heap, or cactus environment, is implemented as a heap containing ⟨closure, environment pointer⟩ structs. When implemented on an instruction machine, each cell in the heap takes 3 machine words.

4.1.1 Compilation

The three instructions are given below, with descriptions of their behavior.

- `TAKE`: Pops a context item off the stack. If the item is an update u, the instruction updates the location u with the current closure. If it is an argument c, the instruction binds the closure c to a fresh location in the cactus environment.
- `ENTER i`: Enters the closure defined by variable index `i`, the current environment pointer, and the current cactus environment.
- `PUSH m`: Pushes the code location `m` along with the current environment pointer.

Each of these instructions corresponds directly to a lambda term: abstraction compiles to `TAKE`, application to `PUSH`, and variables to `ENTER`. Each is compiled using a direct implementation of the transition functions of the $\mathscr{CE}$ machine. The mapping from lambda terms can be seen in Figure 4.1, which defines the compiler. Unlike the TIM, our version of `TAKE` doesn't have an arity; it compiles a sequence of lambdas as a sequence of `TAKE` instructions. Similarly, the `ENTER i` instruction can be implemented either as a loop or unrolled, depending on `i`, and more performance comparisons are needed to determine the trade-off between code size and speed.

The compiler targets x86_64 assembly. Each of the three instructions is mapped onto x86_64 instructions with a macro. The `PUSH` instruction is particularly simple, consisting of only two x86_64 instructions (two `push`es, one for the code pointer and one for the environment pointer). This is an important point: *thunk creation is only two hardware instructions, regardless of environment size.*

$$
\begin{aligned}
C[\![t\ t']\!] &= \mathsf{PUSH}\ label_{C[\![t']\!]} : C[\![t]\!] +\!\!+ C[\![t']\!] \\
C[\![\lambda t]\!] &= \mathsf{TAKE} : C[\![t]\!] \\
C[\![i]\!] &= \mathsf{ENTER}\ i
\end{aligned}
$$

Figure 4.1: $\mathscr{CE}$ machine compilation scheme. C compiles a sequence of instructions from a term. The *label* represents a code label: each instruction is given a unique label. The : operator denotes prepending an item to a sequence and $+\!\!+$ denotes concatenating two sequences.

Syntax

$$t ::= i \mid \lambda t \mid t\ t \mid n \mid op \qquad \text{(Term)}$$
$$n \in \mathbb{I} \qquad \text{(Integer)}$$
$$op ::= + \mid - \mid * \mid / \mid = \mid > \mid < \qquad \text{(PrimOp)}$$
$$v ::= \lambda t[l] \mid n[l] \qquad \text{(Value)}$$

Integer and Primop Semantics

$$\langle n[l], \sigma\ c, \mu, k\rangle \rightarrow \langle c, \sigma\ n[l], \mu, k\rangle \qquad \text{(Int)}$$
$$\langle op[l], \sigma\ n'\ n, \mu, k\rangle \rightarrow \langle op(n', n)[l], \sigma, \mu, k\rangle \qquad \text{(Op)}$$

Figure 4.2: Extensions to the syntax and semantics of the small-step $\mathscr{CE}$ semantics.

4.1.2 Machine Literals and Primitive Operations

Following Sestoft [38], this section describes the extension of the $\mathscr{CE}$ machine to include machine literals and primitive operations. Figure 4.2 shows the parts of syntax and semantics that are new or modified.

For a full definition of the language, see Appendix A. Data types are a common extension which we omit [34, 8]; we take the approach of Sestoft [38], namely that data types can be efficiently implemented with pure lambda terms. For example, consider a list data type (in Haskell syntax): `data List a = Cons a (List a) | Nil`. In an untyped setting, this can be represented in pure lambda terms with $Cons = \lambda h.\lambda t.\lambda c.\lambda n.c\ h\ t$ and $Nil = \lambda c.\lambda n.n$.

Let bindings are another construct commonly included in functional language compilers, even in the internal representation [8, 34]. Non-recursive let is syntactic sugar for a lambda binding and application, and is treated as such. This approach helps ensure that arbitrary lambda terms can be compiled without pre-processing, while other approaches generally require pre-processing [38, 16].

Recursive let bindings are a third omission. Following Rozas [36]: if it can be represented in pure lambda terms, it should be. Recursion is implemented using the standard Y combinator. In the case of mutual recursion, the Y combinator is used in conjunction with a Church tuple of the mutually recursive functions. Without the appropriate optimizations [36], this approach has high overhead, as discussed in Section 4.2.1.

4.1.3 Optimizations

The $\mathscr{CE}$ implementation described in the previous section is completely unoptimized. For example, no effort is expended to discover global functions to avoid costly jumps to pointers in the heap [34]. Indeed, every variable reference will look up the code pointer in the shared environment and jump to it. There is also no implementation of control flow analysis as used by Rozas to optimize away the Y combinator. Thus, every recursive call exhibits the large overhead involved in re-calculating the fixed point of the function.

I do, however, implement two basic optimizations, primarily to reduce the load on the heap:

- `POP`: A `TAKE` instruction can be converted to a `POP` instruction that throws away the operand on the top of the stack if there are no variables bound to the λ term in question. For example, the function $\lambda x.\lambda y.x$ can be implemented with `TAKE`, `POP`, `ENTER 0`.

- `ENTERVAL`: An `ENTER` instruction, when entering a closure that is already a value, should not push an update marker onto the stack. This shortcut prevents unnecessary writes to the stack and heap [34, 17, 38].

4.1.4 Garbage Collection

I have implemented a simple mark and sweep garbage collector with the property that it does not require two spaces, thanks to constant-sized closures in the heap allowing a linked-list representation for the free cells.

Because the focus of this dissertation is not on the performance of garbage collection, we ensure the benchmarks in Section 4.2 are not dominated by GC time.

4.2 Performance Evaluation

This section reports experiments that assess the strengths and weaknesses of the $\mathscr{CE}$ machine. We evaluate using benchmarks from the `nofib` benchmark suite. Because we have implemented only machine integers, and must translate the examples by hand, we use a subset of the `nofib` suite that excludes floating point values and arrays. A list of the benchmarks used and a brief description is given in Figure 4.3.

We compare the $\mathscr{CE}$ machine with two existing implementations:

- GHC: The Glasgow Haskell compiler: A high performance, optimizing compiler

- **exp3:** A Peano arithmetic benchmark. Computes 3^8 and prints the result.
- **queens:** Computes the number of solutions to the nqueens problem for an n by n board.
- **primes:** A simple primes sieve that computes the nth prime.
- **digits-of-e1:** A calculation of the first n digits of e using continued fractions.
- **digits-of-e2:** Another calculation of the first n digits of e using an infinite series.
- **fib:** Naively computes the nth Fibonacci number.
- **fannkuch:** Counts the number of reverses of a subset of a list.
- **tak:** A synthetic benchmark involving basic recursion.

Figure 4.3: Description of Benchmarks

based on the STG machine [34]

- UHC: The Utrecht Haskell compiler: An optimizing compiler based on the GRIN machine [8, 15]

We use GHC version 7.10.3 and UHC version 1.1.9.3. We compile with -O0 and -O3, and show the results for both. Where possible, we pre-allocate a heap of 1GB to avoid measuring the performance of GC implementations. The tests were run on an Intel(R) Xeon(R) CPU E5-4650L at 2.60GHz, running Linux version 3.16.

4.2.1 Results

Figure 4.4 gives the benchmark results. In general, the compiler is outperformed by GHC, sometimes significantly, and outperforms UHC. We spend the remainder of the section analyzing these performance differences.

	$\mathscr{CE}$	GHC -O0	UHC -O0	GHC -O3	UHC -O3
`exp3 8`	1.530	1.176	3.318	1.038	2.286
`tak 16 8 0`	0.366	0.146	1.510	0.006	1.416
`primes 1500`	0.256	0.272	1.518	0.230	1.532
`queens 9`	0.206	0.050	0.600	0.012	0.598
`fib 35`	2.234	0.872	10.000	0.110	8.342
`digits-of-e1 1000`	3.576	1.274	21.938	0.118	22.010
`digits-of-e2 1000`	0.404	0.792	3.430	0.372	3.278
`fannkuch 8`	0.560	0.084	2.184	0.048	2.196

Figure 4.4: Machine Literals Benchmark Results. Measurement is wall clock time, units are seconds. Times averaged over 5 runs ($\sigma < 20\%$).

There are many optimizations built into the abstract machine underlying GHC, but profiling indicates that three in particular lead to much of the performance disparity:

- **Register allocation:** The $\mathscr{CE}$ machine has no register allocator. In contrast, by passing arguments to functions in registers, GHC avoids much heap thrashing.

- **Unpacked literals:** This allows GHC to keep machine literals without tags in registers for tight loops. In contrast, the $\mathscr{CE}$ machine operates entirely on the stack, and has a code pointer associated with every machine literal.

- **Y combinator:** Because recursion in the $\mathscr{CE}$ machine is implemented with a Y combinator, it performs poorly. This could be alleviated with control flow analysis techniques, similar to those used by Rozas [36].

Lack of register allocation is the primary current limitation of the $\mathscr{CE}$ machine. The STG machine pulls the free variables into registers, allowing tight loops with effective register allocation. However, it is less clear how to effectively allocate registers in a fully shared environment setting. That said, it is possible that being lazier

	$\mathscr{CE}$	GHC -O0	UHC -O0	GHC -O3	UHC -O3
`tak 14 7 0`	1.610	2.428	7.936	1.016	7.782
`primes 32`	0.846	1.494	4.778	0.666	5.290
`queens 8`	0.242	0.374	1.510	0.154	1.508
`fib 23`	0.626	0.940	5.026	0.468	5.336
`digits-of-e2 6`	0.138	1.478	5.056	0.670	5.534
`fannkuch 7`	0.142	0.124	0.796	0.040	0.808

Figure 4.5: Church Numeral Benchmark Results. Measurement is wall clock time, units are seconds. Times averaged over 5 runs ($\sigma < 20\%$).

	$\mathscr{CE}$	GHC -O0	UHC -O0	GHC -O3	UHC -O3
`pow 3 8`	0.564	1.994	4.912	0.906	4.932

Figure 4.6: Church Numeral Exponentiation Benchmark Results. Measurement is wall clock time, units are seconds. Times averaged over 5 runs ($\sigma < 20\%$).

about register allocation, e.g., not loading values into registers that may not be used, could have some performance benefits.

To isolate the effect of register allocation and unpacked machine literals, machine integers are replaced with Church numerals in a compatible subset of the evaluation programs. Figure 4.5 shows the performance results with this modification, with the $\mathscr{CE}$ machine occasionally even outperforming optimized GHC.

Next, we consider the disparity due to the Y-combinator, by running a simple exponentiation example with Church numerals, calculating $3^8 - 3^8 = 0$. In this case, the $\mathscr{CE}$ machine significantly outperforms both GHC and UHC, as seen in Figure 4.6
.

These results give us confidence that by adding the optimizations mentioned above, among others, the $\mathscr{CE}$ machine has the potential to be the basis of a real-world compiler. Section 4.3 discusses how some of these optimizations can be applied to the $\mathscr{CE}$ machine.

4.2.2 The Cost of the Cactus

Recall that variable lookup is linear in the index of the variable, following pointers until the index is zero. As one might guess, the lookup cost is high. For example, for the queens benchmark without any optimizations, variable lookup took roughly $80-90\%$ of the $\mathscr{CE}$ machine runtime, as measured by profiling. Much of that cost was for lookups of known combinators, however, so for the benchmarks above, the inlining mentioned in the previous section was incorporated. Still, even with this simple optimization, variable lookup takes roughly 50% of execution time. There is some variation across benchmarks, but this is a rough approximation for the average cost. Section 4.3 discusses how this cost could be addressed in future work.

4.3 Discussion and Related Work

This section compares the compiler presented in this chapter with existing work, and discusses areas for future work.

4.3.1 Closure Representation

Appel and Shao [39] and Appel and Jim [2] both cover the design space for closure representation, and develop an approach called *safely linked closures.* The approach uses flat closures when there is no duplication, and links in a way that preserves liveness, to prevent violation of the *safe for space complexity* (SSC) rule [39]. While we do not address SSC or garbage collection in general, understanding the relationship between SSC and shared environment call-by-need is an interesting area for future work. In particular, hot environments with no sharing could benefit greatly from replacing shared structure with flat.

4.3.2 Eval/Apply vs. Push/Enter

Marlow and Peyton Jones describe two approaches to the implementation of function application: eval/apply, where the function is evaluated and then passed the necessary arguments, and push/enter, where the arguments are pushed onto the stack and the function code is entered [29]. They conclude that despite push/enter being a standard approach to lazy machines, eval/apply performs better. While our current approach uses push/enter, investigating whether eval/apply could be usefully implemented for a shared environment machine like the $\mathscr{CE}$ machine is an interesting avenue for future work.

4.3.3 Collapsed Markers

Friedman et al. show how a machine can be designed to prevent multiple adjacent update markers being pushed onto the stack [17]. This property is desirable because multiple adjacent update markers are always updated with the same value. They give examples showing that in some cases, these redundant update markers can cause an otherwise constant-space stack to overflow. They implement an optimization that collapses update markers by adding a layer of indirection between heap locations and closures. A similar approach, but without the performance hit caused by an extra layer of indirection should be possible, as follows: upon a variable dereference the $\mathscr{CE}$ machine checks if the top of the stack is an update. If it is, instead of pushing a redundant update marker onto the stack, the machine replaces the closure in the heap at the marker location with an update marker pointing to the location specified by the marker on the top of the stack. Then, the variable dereference rule checks if there is an update marker instead of a closure in the dereferenced cell, and if there is, then the value closure pointed to by that update marker will be copied, overwriting the update marker. This effectively makes the update mechanism lazier, only updating one marker eagerly, and any equivalent markers on demand. We leave this optimization for future work.

4.3.4 Register Allocation

One advantage of flat environments is that register allocation is straightforward [1, 34, 44]. It is less obvious how to do register allocation with the $\mathscr{CE}$ machine.

One possible approach that could work well with our shared environment approach would be to only load free variables into registers that are statically known to be needed. In other words, some environment variables may not be used, so only those that will definitely be used should be loaded into registers when a closure is entered, while the rest could be loaded on demand from memory.

4.3.5 Characterizing Performance

While we have provided a few benchmarks and some intuition for why the shared environments would be preferable in some situations, we haven't really characterized when programs will benefit from the approach. This, along with the joining of shared and flat environments, is left for future work. It will likely require a combination of careful performance profiling, static analysis tools, and a deeper understanding of performance tradeoffs.

Chapter 5

Verified Compilation

I don't believe in empirical science.
I only believe in a priori truth.

Kurt Gödel

As discussed in Chapter 1, lazy functional programming languages like Haskell lend themselves particularly well to reasoning about correctness. It is for this reason we are motivated to build a verified compiler for a call-by-need semantics, the default semantics for lazy functional languages: we want this reasoning to be preserved. Unfortunately, one of the challenges for formalization of non-strict compilers is that the semantics of call-by-need abstract machines tend to be complex, incorporating complex optimizations into the semantics, requiring preprocessing of terms, and closures of variable sizes [34, 16]. In this chapter, we use the $\mathscr{CE}$ machine, taking advantage of its simplicity to ease the formal reasoning burden that goes with building a verified compiler.

Verified compilers provide powerful guarantees about the code they generate and its relation to the corresponding source code [10, 28, 24]. In particular, for higher order functional languages, they ensure that the non-trivial task of compiling lambda calculus and its extensions to machine code is implemented correctly, preserving source semantics. The return on investment for verified compilers is high: any reasoning about any program which is compiled with a verified compiler is provably preserved.

Existing verified compilers have focused on call-by-value semantics [10, 28, 24]. This semantics has the property of being historically easier to implement than call-by-need, and therefore likely easier to reason about formally. This chapter formalizes the $\mathscr{CE}$ machine described in Chapter 2. The Coq proof assistant [6] is used to implement and prove the correctness of our compiler. We start with a source language of λ calculus with de Bruijn indices:

$$t ::= t\,t \mid x \mid \lambda\, t$$
$$x \in \mathbb{N}$$

The source semantics is the big-step operational semantics of the $\mathscr{CE}$ machine, which uses shared environments to share results between instances of a bound variable. To strengthen the result, and relate it to a better-known semantics, we also show that the call-by-name $\mathscr{CE}$ machine implements Curien's call-by-name calculus of closures.

It may surprise the reader to see that we do not start with a better known call-by-need semantics; this concern is addressed in Section 5.6. The proof of compiler correctness, along with the proof that our call-by-name semantics implements Curien's semantics, hopefully convinces the reader that we have indeed implemented a call-by-need semantics, despite not using a better known definition of call-by-need.

For our target, we define a simple instruction machine, described in Section 5.3. This simple target allows us to describe the compiler and proofs concisely for the chapter, while still allowing flexibility in eventually verifying a compiler down to machine code for some set of real hardware, e.g., x86, ARM, or Power.

Our main result is a proof that whenever the source semantics evaluates to a value, the compiled code evaluates to the same value. While there are stronger definitions of what qualifies as a verified compiler, We argue that this is sufficient in Section 5.6. This main result, along with the proof that the call-by-name version of our semantics implements Curien's calculus of closures, are the primary contributions of this chapter.

The chapter is structured as follows. In Section 5.1 we describe the source syntax and semantics (the big-step $\mathscr{CE}$ semantics) in detail. We also use this section to define a call-by-name version of the semantics, and show that it implements Curien's calculus of closures [11]. In Section 5.2 we describe the small-step $\mathscr{CE}$ semantics and its relation to the big-step semantics. In Section 5.3 we describe the instruction machine syntax and semantics. In Section 5.4 we describe the compilation from machine terms to assembly language. In Section 5.5 we describe how the evaluation of compiled programs is related to the small-step $\mathscr{CE}$ semantics. We compose this proof with the proof that the small-step semantics implement the big-step semantics to show that the instruction machine implements the big-step semantics. In Section 5.6 we discuss threats to validity, future work, and related work. The Coq source code with all the definitions and proofs described in this chapter is available at `https://github.com/stelleg/cem_coq`.

5.1 Big-Step $\mathscr{CE}$

This section reviews our big-step source semantics. A big-step semantics has the advantage of powerful, easy-to-use induction properties. This eases reasoning about many program properties. We will also review the small-step semantics and prove that it implements the big-step semantics, but by showing that our implementation preserves the big-step semantics, we prove preservation of any inductive reasoning on the structure of evaluation tree.

As in Chapter 3, our source syntax is the lambda calculus with de Bruijn indices. De Bruijn indices count the number of intermediate lambdas between the occurrence of the variable and its binding lambda.

$$t ::= t\ t \mid x \mid \lambda\ t$$
$$x \in \mathbb{N}$$

The essence of the $\mathscr{CE}$ semantics (Figure 3.2) is that it implements a shared environment, and uses its structure to share results of computations. This makes possible a simple abstract machine that operates on the lambda calculus directly, which is uncommon among call-by-need abstract machines [34, 26, 16, 22]. This simplifies formalization, as we do not need to prove that intermediate transformations, e.g., lambda lifting, are semantics-preserving. Another advantage of the $\mathscr{CE}$ machine is that it has constant-sized closures, obviating the need to reason about re-allocating the results of computation and adding indirections required because of closure size changes from thunk to value [34]. We operate on closures, which combine terms with pointers into the shared environment, which is implemented as a heap. Every heap location contains a cell, which consists of a closure and a pointer to the next environment location, which we will refer to as the environment continuation. Variable dereferences index into this shared environment structure, and if/when a dereferenced location evaluates to a value, the original closure (potentially a thunk or closure not evaluated to WHNF) will be replaced with that value. The binding of a new variable extends the shared environment structure with a new cell. This occurs during application, which evaluates the left hand side to an abstraction, then extends the environment with the argument term closed under the environment pointer of the application. The App rule ensures that two variables bound to the same argument closure will point to the same location in the shared environment. Because they point to the same location by construction of the shared environment, we can update that location with the value computed at the first variable dereference, and then each subsequent dereference will point to this value. The variable rule applies the update by indexing into the shared environment structure and replacing the closure at that location with the resulting value. It is worth noting that while the closures in the heap cells are mutable, the shared environment structure is never mutated. This property is crucial when reasoning about variable dereferences. The $\mu(l,i)$ function looks up a variable index in the shared environment structure by following environment continuation pointers, returning the location and cell pointed to by the final step. See the Coq source for a formal treatment. Note that we require that fresh heap locations are greater than zero. This is required for reasoning about compilation to the instruction machine, which we will return to in Section 5.3. While here we constrain fresh heap locations to be fresh with respect to the entire heap domain, for a real implementation, this is far too strong a constraint, as it doesn't allow any sort of heap re-use. We return to this issue in Section 5.6, and discuss how this could be relaxed to either allow reasoning about garbage collection or direct heap reuse.

The fact that our natural semantics is defined on the lambda calculus with de Bruijn indices differs from most existing definitions of call-by-need, such as Ariola's call-by-need [4] or Launchbury's lazy semantics [26]. These semantics are defined on the lambda calculus with named variables. While it should be possible to relate our semantics to these, the comparison is made more difficult by this disparity.[1] A more fruitful relation to semantics operating on the lambda calculus with named variables would likely be relating Curien's calculus of closures to call-by-name semantics implemented with substitution. We return to this discussion in Section 5.6.

As mentioned in Section 3, these big-step semantics do not explicitly include a notion of nontermination. Instead, nontermination would be implied by the negation of the existence of an evaluation relation. This prevents reasoning directly about

[1]Both of these well known existing semantics have known problems that arise during formalization, as discussed in Section 5.6.

nontermination in an inductive way, but for the purpose of our primary theorem this is acceptable.

One interesting property of defining an inductive evaluation relation in a language such as Coq is that computation can be done on the evaluation tree. In other words, the evaluation relation given above defines a data type, one that computation can be done on in standard ways. For example, we could potentially compute properties such as size and depth, which would be related to operational properties of compiled code.

Finally, given a term t, the initial configuration is defined as $(t\,[0]\,,\varepsilon)$. As discussed, the choice of the null pointer for the environment pointer is not completely arbitrary, but chosen across our semantics uniformly to represent failed environment lookup.

5.1.1 Call-By-Name

This section reviews the call-by-name variant of our big-step semantics and provides a proof that it is an implementation of Curien's call-by-name calculus of closures.

See Figure 3.1 for the definition of our call-by-name semantics. Note that the only change from our call-by-need semantics is that we do not update the heap location with the result of the dereferenced computation. This is the essence of the difference between call-by-name and call-by-need.

Recall Figure 2.1 for a formalization of Curien's call-by-name semantics. We define a heterogeneous equivalence relation between our shared environment and Curien's environment. Effectively, this relation is the proposition that the shared environment structure is a linked list implementation of the environment list in Curien's semantics. This is defined inductively, and we require that every closure reachable in the environment is also equivalent. We say two closures are equivalent if their terms are identical and their environments are equivalent.

Given these definitions, we can prove that the call-by-name $\mathscr{CE}$ semantics implements Curien's call by name semantics:

Theorem 1. *If a closure* c *in Curien's call-by-name semantics is equivalent to a configuration* c'*, and* c *steps to* v*, then there exists a* v' *that our call-by-name semantics steps to from* c' *that is equivalent to* v*.*

Proof outline. The proof proceeds by induction on Curien's step relation. The abstraction rule is a trivial base case. The variable lookup rule uses a helper lemma that proves by induction on the variable that if the two environments are equivalent and the variable indexes to a closure, then the μ function will look up an equivalent closure. The application rule uses a helper lemma which proves that a fresh allocation will keep any equivalent environments equivalent, and that the new environment defined by the fresh allocation will be equivalent to the extended environment of Curien's semantics.

By proving that Curien's semantics is implemented by the call-by-name variant of our semantics, we provide further evidence that our call-by-need is a meaningful semantics.

One important note is that nowhere do we require that a term being evaluated is closed under its environment. Indeed, it's possible that a term with free variables can be evaluated by both semantics to a value as long as a free variable is never dereferenced. This theme will recur through the rest of the chapter, so it is worth keeping in mind.

5.2 Small-Step $\mathscr{CE}$

In this section we review the small-step semantics of the $\mathscr{CE}$ machine, and show that it implements the big-step semantics of Section 5.1. This is a fairly straightforward transformation implemented by adding a stack. The source language is the same, and we simply add a stack to our configuration (and call it a state). The stack elements are either argument closures or update markers. Update markers are pushed onto the stack when a variable dereferences that location in the heap. When they are popped by an abstraction, the closure at that location is replaced by said abstraction, so that later dereferences by the same variable in the same scope dereference the value, and do not repeat the computation. Argument closures are pushed onto the stack by applications, with the same environment pointer duplicated in the current closure and the argument closure. Argument closures are popped off the stack by abstractions, which allocate a fresh memory location, write the argument closure to it, write the environment continuation as the current environment pointer, then enter the body of the abstraction with the fresh environment pointer. This is the mechanism used for extending the shared environment structure. The semantics is defined formally in Figure 3.3.

5.2.1 Relation to Big Step

Here we prove that the small-step semantics implements the big-step semantics of Section 5.1. This requires first a notion of reflexive transitive closure, which we define in the standard way. We also make use of the fact that the reflexive transitive closure can be defined equivalently to extend from the left or right.

Lemma 1. *If the big-step semantics evaluates from one configuration to another, then the reflexive transitive closure of the small-step semantics evaluates from the same starting configuration with any stack to the same value configuration with that* *same stack.*

Proof outline. The proof proceeds by induction on the big-step relation. We define our induction hypothesis so that it holds for all stacks, which gives us the desired case of the empty stack as a simple specialization. The rule for abstractions is the trivial base case. Var rule applies as the first step, and the induction hypothesis applies to the stack with the update marker on it. To ensure that the Upd rule applies we use the fact that the big-step semantics only evaluates to abstraction configurations, and the fact that the reflexive transitive closure can be rewritten with steps on the right. For the Application rule, we take advantage of the fact that one can append two evaluations together, as well as extend a reflexive transitive closure from the left or the right. As with the Var rule we use the fact that the induction rule is defined for all stacks to ensure evaluation of the left hand side to a value with the argument on the top of the stack. Finally, we extend the environment with the argument closure, and evaluate the result to a value by the second induction hypothesis.

Adding a stack in this fashion is a standard approach to converting between big step and small-step semantics, and applies here in a straightforward way.

5.3 Instruction Machine

Here we describe in full the instruction machine syntax and semantics. We choose a simple stack machine with a Harvard architecture (with separate instruction and heap memory). We use natural numbers for pointers, though it shouldn't be too difficult to replace these with standard-sized machine words, e.g., 64 bits, making the stack and malloc operations partial to better represent real-world hardware. Our stack is represented as a list of pointers, though again it should be a relatively straightforward exercise to represent the stack in contiguous memory. We define

$$
\begin{array}{rll}
n, l, w \in \mathbb{N} & & \text{(Machine Word)} \\
r := & ip \mid ep \mid r1 \mid r2 & \text{(Registers)} \\
wo := & r \mid r\%n & \text{(Write Operands)} \\
ro := & wo \mid n & \text{(Read Operands)} \\
i := & push\ ro \mid pop\ wo \mid new\ n\ wo \mid mov\ ro\ wo & \text{(Instructions)} \\
bb := & i : bb \mid jump\,\{ro, l\}\,ro & \text{(Basic Block)} \\
p := & bb* & \text{(Program)} \\
s := & w* & \text{(Stack)} \\
h := & (l, w)* & \text{(Heap)} \\
S := & \langle rf, p, s, h \rangle & \text{(State)}
\end{array}
$$

Figure 5.1: Instruction Machine Syntax

our machine to have only four registers: an instruction pointer, an environment pointer, and two scratch registers. Our instruction set is minimal, consisting only of a conditional jump instruction, pop and push instructions, a move instruction, and an instruction for allocating new memory. Note that for our program memory, we have pointers to basic blocks, but for simplicity of proofs we choose to not increment the instruction pointer within a basic block. Instead, the instruction pointer is constant within a basic block, only changing between basic blocks. In fact, we represent the program as a list of basic blocks, with pointers indexing into the list. This has the advantage of letting us easily reason about sublists and their relation to terms. The full syntax of the machine is given in Figure 5.1. Note that curly brackets {} denote optionality, while stars $*$ denote zero or more elements, represented as a list. Note that we'll use some common list terminology, such as bracket notation for indexing, i.e., $l\,[i]$ accesses the i'th element in l (we don't worry about the partiality in this presentation of this operation; see the Coq implementation for a full jreatment). In addition to the syntax defined in Figure 5.1, we use $+\!\!+$ for list concatenation, and :: for consing an element onto the head of a list.

We separate read (ro) and write (wo) operands. Write operands can be registers or memory (defined by a register and a constant offset). Read operands can be any write operand or a constant. For reading, there is the read relation, which takes a read operand and a state and is inhabited when the third argument can be read from that read operand in that state. Similarly, a write relation is inhabited when writing the second argument into the first in a state defined by the third argument results in the state defined by the fourth argument.

The machine semantics should be fairly unsurprising. A State consists of a register file, program memory, a stack, and a heap. The push instruction takes a read operand and pushes it onto the stack. The pop instruction pops the top of the stack into a write operand. The mov instruction moves a machine word from a read operand to a write operand. The jump instruction is parameterized by an optional pair, which, if present, reads the first element of the pair from a read operand, checks if it is zero, and if so sets the IP to the second element of the pair, which is a constant pointer. If the condition is not zero, then it sets the IP to the instruction pointer contained in the second jump argument. If we pass nothing as the first argument, then it becomes an unconditional set of the IP to the value read from the second argument. Note that the second argument is a read operand, so it can either be a constant or read from a register or memory. This means it can be effectively either a direct or indirect jump, both of which are used in the compilation of lambda terms. The new instruction allocates a contiguous block of new memory and writes the resulting pointer to the fresh memory into a write operand. We take the approach of not choosing a particular allocation strategy. Instead, we follow existing approaches and parameterize our proof on the existence of such functionality [10]. The complete semantics of the machine is given in Figure 5.2. Note that we separate instruction steps and basic block steps. Recall that a basic block is a sequence of instructions that ends with a jump. The Step BB relation will execute the instructions in the basic block in order, then set the IP in accordance with the jump semantics. The

Step relation dereferences a basic block at the current IP, and if executing the basic block results in a new state, then the machine executes to that state.

5.4 Compiler

In this section we describe the compiler, which compiles lambda terms with de Bruijn indices to programs. The compiler proceeds by recursion on lambda terms, keeping a current index into the program to ensure correct linking without a separate pass. For variables, when we get to zero we push the current environment pointer and a null instruction pointer to denote the update marker to the location of the closure being entered. Then we *mov* the closure at that location into $r1$ and ep, and *jump* to $r1$, recalling that the *jump* sets the ip. For nonzero variables, we replicate traversing the environment pointer i times before loading the closure. For applications, we calculate the program location of the argument basic block, and push that and the current environment pointer onto the stack, effectively pushing an argument closure on top of the stack. We then *jump* to the left hand side of the application, as is standard for push-enter evaluation. For abstractions, we use a conditional jump depending on whether the top of the stack is a null pointer (and therefore an update marker) or a valid instruction pointer (and therefore an argument). If it is an update marker, we update the heap location defined by the update marker with the current value instruction pointer and the current environment pointer. We must point to the first of the three abstractions basic blocks, as this value could later update another heap location as well. In the case that the top of the stack was a valid instruction pointer, we allocate a new chunk of 3 word of memory, and *mov* the argument closure into it, with the current environment pointer as the environment continuation. We then set our current environment pointer to this fresh location. This is the process by which we extend our shared environment structure in the instruction machine. Finally, we perform an unconditional jump to the next basic block, which is the first basic block of the compiled body of the lambda. As this is an unconditional jump to the next basic block, for real machine code this jump can be omitted.

Being able to define the full compiler this simply is crucial to this verification project. Other, more sophisticated implementations of call-by-need, such as the STG machine, are much harder to implement and reason about. It is worth noting that despite this simplicity, initial tests suggest that performance is reasonable, and is often competitive with state of the art (Chapter 4).

As with the relation discussed in Section 5.1, a term does not have to be closed to compile it. Indeed, we will happily generate code that if entered, will attempt to dereference the null pointer, leaving the machine stuck. Because we are only concerned with proving that the source semantics are implemented in the case that it evaluates to a value, this is not a problem. If we wanted a more general theorem, we could try to show that if the source semantics gets stuck trying to dereference a free variable, the implementation would get stuck in the same way, both failing to dereference a null pointer.

5.5 Compiler Correctness

In this section we define a relation between the state of the small-step semantics and the state of the instruction machine semantics, and show that the instruction machine implements the small-step semantics under that relation.

In general, we implement closures as instruction pointer, environment pointer pairs. For the instruction pointers, we relate them to terms via the compile function defined in Section 5.4. Essentially, we require that the instruction pointer points to a list of basic blocks that the related term compiles to. For the current closure, we relate the instruction pointer register in the instruction machine to the current term in the small-step source semantics. The environment pointers of each machine are more similar. Given a relation between the heaps of the two machines, we define the relation between two environment pointers as existing in the relation of the heaps, or both being the null pointers. While it should be possible to avoid this special case, during the proof it became apparent that not having the special case made the proof significantly harder. This forces us to add the constraint to all machines that pointers are non-null, which for real hardware shouldn't be an issue.

Null pointers are crucial in two ways. First, they explicitly define the root of the shared environment structure in both the source semantics and the machine semantics. Second, they are used for instruction pointers. To differentiate between update markers and pointers to basic blocks, a null pointer is used to refer to an update marker, and a non-null pointer defines an instruction pointer for an argument closure. Note that in fact, while the null pointers in heaps required us to only allocate non-null fresh locations in the heaps of the $\mathscr{CE}$ semantics, using null pointers to denote update markers requires no change to our program generation, due to the fact that an argument term of an application cannot occur at position 0 in the program.

The relation between the heaps of the small-step source semantics and the instruction machine is the trickiest part of the state relation. Note that for each location in the source semantics heap, we have a cell with a closure and environment continuation pointer. Naturally, the instruction machine represents these as three pointers: two for the closure (the instruction pointer and environment pointer) and one for the environment continuation. The easiest approach turned out to be to use the structure of the heap constructs to define a one-to-three mapping between this single cell and the three machine words. The structure used for each of the heaps is a list of pointer, value bindings. We use the ordering of these bindings in the list to define a one binding to three binding mapping between the source heap and the machine heap. We define a membership relation that defines when an element is in our heap relation, proceeding recursively on the inductive relation structure. This allows us to define a notion of which pairs of each type of closure are in the heap, along with their respective locations. Due to the ordering in which they are allocated in the heap during evaluation, each pair of memory allocations corresponds to an equivalent cell. We use this property as a heap equivalence property that is preserved through evaluation: every binding pair in the heap relation property described above defines equivalent closures and environment continuations. For the relation between our stacks, we define a similar notion. For update markers, we require that every update marker points to related environments (they are two pointers that exist in the heap relation). For argument closures, we require that the closures are equivalent (the instruction pointer and environment pointer are equivalent to their respective counterparts in the small-step semantics).

In summary, we require that the current closure in the small-step semantics is equivalent to the closure represented by the instruction pointer, environment pointer pair, and that the stacks and the heaps are equivalent. The actual Coq implementation of this relation is too involved to relate directly here. We encourage the reader to read the linked Coq source to fully appreciate it.

Given this relation between heaps, we can state the primary lemma.

Lemma 2. *Given that an instruction machine state i is related to a small-step semantics state s, and that small-step semantics state steps to a new state s', the instruction machine will step in zero or more steps to a related state i'.*

Proof outline. Our proof proceeds by case analysis on the step rules for the small-step semantics. We'll focus on the second half of the proof, that i' is related to s'. The proofs that i evaluates to i' follow fairly directly from the compiler definition given in Section 5.4. For the Var rule, because we need to proceed by induction, we have to define a separate lemma and proceed by induction on a basic block while forgetting the program, as the induction hypothesis is invalid in the presence of the program. We then use the lemma to show that evaluation of a compiled variable implements the evaluation of the variable in the small-step semantics. In particular, we use the null environment as a base case for our induction, as we know the only way lookup could fail is if both environment pointers are null, but that cannot be the case due to the fact that we know that the small-step semantics must have successfully looked up its environment pointer in the heap. Therefore the only option is for both environment pointers to exist in the heap relation, which when combined with the heap equivalence relation in the outer proof gives us the necessary property that the environment continuations are equivalent. Finally, because the last locations reached must have been in the heap relation, we know they are equivalent environment pointers, and therefore the stack relation is preserved when we push the update marker onto the heap. For the App rule, we use the definition of our compiler to prove that the argument term and argument instruction pointer are equivalent and that the left hand side term and instruction pointer are also equivalent. They share an environment pointer which is equivalent by the fact that the application closures are related. This proves that the stack relation is preserved as well as the current closure, while the heap is unchanged. For the Lam rule, we allocate a fresh variable and because of our stack relation we can be sure that the closures that we allocate are equivalent, as well as the environment continuations, as they are taken from the previous current continuation. Because of how we define it, the new allocations are equivalent under our heap relation, and preserve heap equivalence. Finally, the Upd rule trivially preserves the stack and current closure relations, and for proving that the relation is preserved for the heap, we proceed by induction on the heap relation. In addition, we must prove a supporting lemma that all environment relations are preserved by the update.

We now have a proof that the small-step semantics implements the big-step semantics, and a proof that the instruction machine implements the small-step semantics. We can now combine these to get our correct compiler theorem.

Theorem 2. *If a term t placed into the initial configuration for the big-step semantics evaluates to a value configuration v, then the instruction machine starting in the initial state with* `compile 0 t` *as its program will evaluate to a related state v'.*

Proof outline. We first require that the relation defined between the small-step semantics state and the instruction machine state holds for the initial configurations. This follows fairly directly from the definition of the initial conditions and the compile function. Second, we have by definition of reflexive transitive closure that Lemma 2 implies that if the reflexive transitive closure of the small-step relation evaluates in zero or more steps from a state c to a state v, then a related state of the instruction machine c' will evaluate to a state v' which is related to v. We use these two facts, along with the proof that the small-step implements the big-step for any stack, specialized on the empty stack, to prove our theorem.

It is worth recalling exactly what the relation implies about the two value states. Namely, in addition to the value closures being equivalent, their heaps and environments are equivalent, so that every reachable closure in the environment is equivalent between the two.

5.6 Discussion

This section reflects on the chapter, including threats to validity, future work, related work, and general discussion of the results.

One thing that is important to communicate is the difficulty of writing comprehensible proofs. The reader is discouraged from attempting to understand the proofs in any way by reading the Coq tactic source code. While I attempted to keep the definitions and lemmas as clean and comprehensible as possible, I found it extremely difficult to do the same with tactics. Partially this may be a failure on my part to become more familiar with the tactic language of Coq, but I suspect that the imperative nature of tactic proofs prevents composability of tactic meta-programs.

Another lesson was the importance of good induction principles. For example, in Section 5.6.1, we discuss the issue of only proving the implication of correctness in one direction. This is effectively a product of the power of the inductive properties of high level semantics, which makes them so much easier to reason about. Indeed, this lesson resonates with the purpose of the chapter, which is that we'd like to reason about high level semantics, because they are so much easier to reason about due to their pleasant inductive properties, and have that reasoning preserved through compilation.

5.6.1 Threats to Validity

There are a few potential threats to validity that we address in this section. The first is the one mentioned in Section 2.4, that we only show that our compiler is correct in the case of termination of the source semantics. In other words, if the source semantics doesn't terminate, the theorem says nothing about how the compiled code behaves. This means that we could have a compiled program that terminates when the source semantics does not terminate.

One argument in defense of verification presented here is that *we generally only care about preservation of semantics for preserving reasoning about our programs.* In other words, if we have a program that we can't reason about, and therefore may not terminate, we care less about having a proof that semantics are preserved. Of course, this is a claim about most uses of program analysis. There are possible analyses that could say things along the lines of *if* the source program terminates, then we can conclude x. These cases are rare, and therefore the provided proof of correctness can still be applied to most use cases.

Another potential threat to validity is the use of a high level instruction machine language. While we claim that its high level and simplicity should make it possible to show that a set of real ISAs implement this instruction machine, we haven't formally verified this step. This would make for valuable future work, and nothing in the design of our high level instruction machine prevents such work.

As a dual to the issue of a high level instruction machine language, some readers may take issue with calling lambda calculus with de Bruijn indices an "input language". Indeed, we do not advocate writing programs in such a language. Still, the conversion between lambda calculus with named variables and lambda calculus with de Bruijn indices is a well understood topic, and it would distract from the presentation of the verified compiler provided here. Indeed, as noted below, semantics using named variables and substitution can be hard to get right [9, 31], so we stand by our decision to use a semantics based on lambda calculus with de Bruijn indices. One potential approach for future work would be to prove that a call-by-name semantics using substitution is equivalent to Curien's calculus of closures, which when combined with a proof that our call-by-need implements our call-by-name, would prove the compiler implements the semantics of a standard lambda calculus with named variables.

A third threat to the validity of this work is the question of whether we have really proved that we have implemented *call-by-need*. The question naturally arises of what exactly it means to prove an implementation of call-by-need is correct. There are certainly well-established semantics [26, 4], so one option would be to directly prove that the $\mathscr{CE}$ semantics implements one of those existing semantics. Unfortunately, recent work has shown that both of these have small issues that arise when formalized that require fixes. Indeed, we did stray down this path and rediscovered one of these issues which has been previously described in the literature [31]. This raises the question of whether or not semantics that aren't obviously correct are a good base for what it means to be a call-by-need semantics. Instead, we have chosen to relate our call-by-name semantics formally to a semantics that is obviously correct, Curien's calculus of closures. Along with the small modification required for memoization of results, we hope that we have convinced the reader that it is *extremely likely* that the memoization of results is correct. Of course, further evidence such as examples of correct evaluation would go further to convince the reader, and for that we encourage readers to play with a toy implementation at `https://github.com/stelleg/cem_pearl`. Finally, a more convincing result would be a proof that the call-by-need semantics implement the call-by-name semantics.

Yet another threat to validity is our approach (or lack of approach) to heap-reuse. For simplicity, we have assumed that our fresh locations are fresh with respect to all existing bindings in the heap. Of course, this is unsatisfactory when compared to real implementations. It would be preferable to have our freshness constraint relaxed to only be fresh with respect to live bindings on the heap. This modification should be possible, at the cost of increased complexity in the proofs.

5.6.2 Future Work

In addition to some of the future work discussed as ways of addressing issues in Section 5.6.1, there are some additional features that we think make for exciting areas of future work.

One such area is reasoning about preservation of operational properties such as time and space requirements. This would enable reasoning about time and space properties at the source level and ensuring that these are preserved through compilation. In addition, there is the possibility of verified optimizations, where one can prove that some optimizations are both *correct*, in that they provably preserve semantics, and *true* optimizations, in that they only improve performance with respect to some performance model. By defining a baseline compiler and proving that it preserved operational properties such as time and space usage, one would have a good platform for which to apply this class of optimizations, resulting in a full compiler that verifiably preserves bounds on time and space consumption. As with correctness, reasoning about operational properties is often likely to be easier in the context of the easy-to-reason-about high level semantics, and having that reasoning provably preserved would be extremely valuable.

Another exciting area of future work is powerful proofs of type preservation through compilation. While there has been existing work on type-preserving compilers, fully verified compilers like this one provide such a strong property that type-safety should fall out directly.

One useful feature of Coq is the ability to extract Coq programs out to other implementations, e.g., Haskell. This raises the possibility of extracting the verified compiler out to a Haskell implementation that could be incorporated into GHC, providing a path towards a verified Haskell compiler.

5.6.3 Relation to Ariola et al.

As mentioned above, to improve the relation to existing work, we could relate the semantics to the operational semantics of Ariola et al., seen in Figure 5.4 [4]. Because I spent a large amount of time attempting this, I use this section to discuss challenges to this approach.

One approach I tried was to show bisimulation between $\rightarrow_N$ and $\Downarrow$. Unfortunately, as mentioned in Section 5.6.1, there is an issue I discovered when formalizing this semantics. Essentially, the rules for variable freshness are insufficient. To fix this, one has to modify the rules to ensure global freshness. This modification is shown in Figure 5.5. In addition to this complication, the complexity of managing the relation between de Bruijn terms and standard named terms through a mutating heap makes this proof extremely challenging.

5.6.4 Related Work

Chlipala implements a compiler from a STLC to a simple instruction machine in [10]. In many ways it is more sophisticated than our work: it converts to CPS, performs closure conversion, and proves a similar compiler correctness theorem to the one we've proved here. The primary difference is that we've defined a call-by-need compiler, which forces us to reason about updating thunks in the heap, a challenge not faced by call-by-value implementations.

Breitner formalizes Launchbury's natural semantics and proves an optimization is sound with respect to the semantics [26, 9]. By relating his formalization with ours, these projects could be combined to prove a more sophisticated lazy compiler correct: one with non-trivial optimizations applied.

CakeML [24] is a verified compiler for a large subset of the Standard ML language formalized in HOL4 [40]. Like Chlipala's work, this is a call-by-value language, though they prove correctness down to an x86 machine model, and are working with a much larger real-world source language. They also make divergence arguments along the lines of [32], strengthening their correctness theorem in the presence of nontermination. It's also worth noting that like [28], they are also formalizing a front end to the compiler.

As part of the DeepSpec project, Weirich et al. have been working on formalizing Haskell's core semantics [48, 41]. There is opportunity to use my work in combination with the DeepSpec project to implement and verify a full-featured Haskell compiler.

$$\frac{\begin{array}{c} \mathit{read}\ \mathit{ro}\ \langle \mathit{rf}, \mathit{ps}, h\rangle\ v \\ \mathit{bb}, \langle \mathit{rf}, p, v :: s, h\rangle \rightarrow_{\mathit{bb}} S \end{array}}{\mathit{push}\ \mathit{ro} : \mathit{bb}, \langle \mathit{rf}, p, s, h\rangle \rightarrow_{\mathit{bb}} S} \tag{Push}$$

$$\frac{\begin{array}{c} \mathit{write}\ \mathit{wo}\ w \langle \mathit{rf}, p, s, h\rangle S' \\ \mathit{bb}, S' \rightarrow_{\mathit{bb}} S \end{array}}{\mathit{pop}\ \mathit{wo} : \mathit{bb}, \langle \mathit{rf}, p, w :: s, h\rangle \rightarrow_{\mathit{bb}} S} \tag{Pop}$$

$$\frac{\begin{array}{c} \forall i < n, f + i \notin \mathsf{dom}\,(h) \\ \mathit{write}\ \mathit{wo}\ f \langle \mathit{rf}, p, s, \mathit{zeroes}\ n\ f \mathbin{+\!\!+} h\rangle S' \\ \mathit{bb}, S' \rightarrow_{\mathit{bb}} S \end{array}}{\mathit{new}\ n\ \mathit{wo} : \mathit{bb}, \langle \mathit{rf}, p, s, h\rangle \rightarrow_{\mathit{bb}} S} \tag{New}$$

$$\frac{\mathit{read}\ \mathit{ro}\ s\ v \quad \mathit{write}\ \mathit{wo}\ v\ S\ S' \quad \mathit{bb}, S' \rightarrow_{\mathit{bb}} S''}{\mathit{mov}\ \mathit{ro}\ \mathit{wo} : \mathit{bb}, S \rightarrow_{\mathit{bb}} S''} \tag{Mov}$$

$$\frac{\mathit{read}\ \mathit{ro}\ S\ 0 \quad \mathit{write}\ \mathit{ip}\ k\ S\ S'}{\mathit{jump}\ (\mathit{ro}, k)\ \ j, S \rightarrow_{\mathit{bb}} S'} \tag{Jump 0}$$

$$\frac{\begin{array}{c} l > 0 \quad \mathit{read}\ \mathit{ro}\ S\ l \\ \mathit{read}\ j\ S\ k \quad \mathit{write}\ \mathit{ip}\ k\ S\ S' \end{array}}{\mathit{jump}\ (\mathit{ro}, k')\ \ j, S \rightarrow_{\mathit{bb}} S'} \tag{Jump S}$$

$$\frac{\mathit{read}\ \mathit{ro}\ S\ l \quad \mathit{write}\ \mathit{ip}\ l\ S\ S'}{\mathit{jump}\ \mathit{ro} : S \rightarrow_{\mathit{bb}} S'} \tag{Jump}$$

$$\frac{\begin{array}{c} \mathit{read}\ \mathit{ip}\ \langle \mathit{rf}, p, s, h\rangle\ k \\ p\,[k] = \mathit{bb} \\ \mathit{bb}, \langle \mathit{rf}, p, s, h\rangle \rightarrow_{\mathit{bb}} S' \end{array}}{\langle \mathit{rf}, p, s, h\rangle \rightarrow S'} \tag{Enter}$$

Figure 5.2: Instruction Machine Semantics

$$
\begin{aligned}
var\ 0 &:= push\ ep : \\
&\quad push\ 0 : \\
&\quad mov\ (ep\%0)\ r1 : \\
&\quad mov\ (ep\%1)\ ep : \\
&\quad jump\ r1 \\
var\ (i+1) &:= mov\ (ep\%2)\ ep : \\
&\quad var\ i \\
compile\ i\ k &:= [var\ i] \\
compile\ (m\ n)\ k &:= let\ ms = compile\ m\ (k+1)\ in \\
&\quad let\ nk = 1+k+length\ ms\ in \\
&\quad push\ ep : \\
&\quad push\ nk : \\
&\quad jump\ (k+1) :: \\
&\quad ms +\!\!+ compile\ n\ nk \\
compile\ (\lambda b)\ k &:= pop\ r1 : \\
&\quad jump\ (r1, k+1)\ (k+2) :: \\
&\quad pop\ r1 : \\
&\quad mov\ k\ r1\%0 : \\
&\quad mov\ ep\ r1\%1 : \\
&\quad jump\ k :: \\
&\quad new\ 3\ r2 : \\
&\quad mov\ r1\ (r2\%0) : \\
&\quad pop\ (r2\%1) : \\
&\quad mov\ ep\ (r2\%2) : \\
&\quad mov\ r2\ ep : \\
&\quad jump\ (k+3) :: \\
&\quad compile\ b\ (k+3)
\end{aligned}
$$

Figure 5.3: Compiler Definition

$$\Phi, \Psi, \Upsilon ::= x_1 \mapsto t_1, \ldots, x_n \mapsto t_n \tag{Heap}$$

$$\frac{\langle\Phi\rangle t \Downarrow \langle\Psi\rangle \lambda x.t'}{\langle\Phi, x \mapsto t, \Upsilon\rangle x \Downarrow \langle\Psi, x \mapsto \lambda x.t', \Upsilon\rangle \lambda x.t'} \tag{Id}$$

$$\frac{}{\langle\Phi\rangle \lambda x.t \Downarrow \langle\Phi\rangle \lambda x.t} \tag{Abs}$$

$$\frac{\begin{array}{c}\langle\Phi\rangle t_l \Downarrow \langle\Psi\rangle \lambda x.t_n \\ \langle\Psi, x' \mapsto t_m\rangle [x'/x]t_n \Downarrow \langle\Upsilon\rangle \lambda y.t'\end{array}}{\langle\Phi\rangle t_l\ t_m \Downarrow \langle\Upsilon\rangle \lambda y.t'} \tag{App}$$

Figure 5.4: Ariola et al.'s Operational Semantics

$$\Phi, \Psi, \Upsilon ::= x_1 \mapsto t_1, \ldots, x_n \mapsto t_n \tag{Heap}$$

$$\frac{\langle\Phi, x \mapsto t, \Upsilon\rangle t \Downarrow \langle\Phi', x \mapsto t, \Upsilon'\rangle \lambda x.t'}{\langle\Phi, x \mapsto t, \Upsilon\rangle x \Downarrow \langle\Phi', x \mapsto \lambda x.t', \Upsilon'\rangle \lambda x.t'} \tag{Id}$$

$$\frac{}{\langle\Phi\rangle \lambda x.t \Downarrow \langle\Phi\rangle \lambda x.t} \tag{Abs}$$

$$\frac{\begin{array}{c}\langle\Phi\rangle t_l \Downarrow \langle\Psi\rangle \lambda x.t_n \\ \langle\Psi, x' \mapsto t_m\rangle [x'/x]t_n \Downarrow \langle\Upsilon\rangle \lambda y.t'\end{array}}{\langle\Phi\rangle t_l\ t_m \Downarrow \langle\Upsilon\rangle \lambda y.t'} \tag{App}$$

Figure 5.5: Ariola et. al's Operational Semantics (Fixed)

Chapter 6

Conclusions

Understand as well as I may, my comprehension can only be an infinitesimal fraction of all I want to understand.

Ada Lovelace

This dissertation is a thorough investigation of a shared-environment approach to implementing call-by-need semantics. Chapter 4 investigated the runtime efficiency advantages by implementing a simple native code compiler. Despite the compiler's lack of optimization framework, it was often competitive with the state of the art. From this, the conclusion is that this approach is a promising abstract machine for real-world compilers. Chapter 5 showed how to use the simplicity of the implementation to effectively reason formally about its correctness. While it was a significant undertaking, the success of the verified compiler provides strong evidence that the simplicity of the machine is a valuable property, and that property can be further exploited in future work.

For the rest of this Chapter, we take a retrospective look at the work done for the dissertation, discussing both what worked well and what didn't. The hope is that this deeper dive into the challenges and successes throughout the dissertation can better inform future work, both work that uses the $\mathscr{CE}$ machine directly, and work that might take a different approach, but with similar goals. While some of what is discussed in this section has been covered in previous Chapters, we try and address big-picture conclusions here, accumulating the lessons learned along the way. In addition, through discussions with other experts and through further introspection, more conclusions and lessons learned have been discovered, and we use this section to bring those to light.

More specifically, this chapter attempts to do a few things. First, it summarizes the theses and conclusions of the dissertation. Second, it summarizes and addresses the threats to validity of the conclusions presented. Largely motivated by addressing these threats to validity, it then discusses future directions enabled by this work.

6.1 Threats to Validity

This section attempts to summarize and address the most pressing threats to the validity to the theses and conclusions discussed in this dissertation. While it attempts to enumerate the most pressing threats, any list of this nature will be incomplete, and therefore the goal is not to create a complete list. Instead, the aim of the section is simply to convey that possible criticisms have been seriously considered. Note that Chapters 4 and 5 have chapter-specific threats to validity, so in this section we instead focus on more general threats to validity.

The first, and most glaring, is that the verified compiler makes no claims about the behaviour of the compiled code in the case of non-termination in the source semantics. One point to make is that even if we accept that this work only applies to total languages, and they do exist (Agda, Coq, STLC, etc.) [45], then in the context of those languages, this is a complete verified compiler.

Some readers may be inclined to dismiss a compiler, like the one presented here, that doesn't implement recursion and algebraic data types explicitly. A tangential goal of this dissertation is to open the reader to the possibility that those are not necessary for high performance code. Removing them certainly makes formal reasoning simpler, something that hopefully the reader is convinced is an important property for a compiler to have.

6.2 Future Work

While Chapters 4 and 5 have sections dedicated to future work specific to their topics, we expand on those here, discussing future work that could combine the approaches of both efforts.

The most obvious one would be to combine the efforts. By extracting the Coq implementation of the compiler into Haskell, the language that the native code compiler is implemented in, we would attain a verified fragment of a true native code compiler. One could then work towards extending the proof to cover more of the implementation over time. When combined with the possibility of implementing a full Haskell compiler using the native code compiler in this work, we have a viable path towards a high performance, fully verified compiler for Haskell.

This approach neglects the fact that there are many cases where GHC significantly outperforms the native code compiler in its current form. Recent work on verified transformations and optimizations, when combined with our verified compiler, could result in a verified compiler that is actually competitive with GHC in performance, likely outperforming it in some cases due to lightweight closure creation enabled by the $\mathscr{CE}$ machine. In a sense, this would be a natural next step to the existing type-safe core language of GHC: instead of just ensuring that transformations are type-safe, we could have a compiler that ensures that they are correct.

While discussed briefly in the previous chapters, one exciting area for future work is reasoning formally about performance. Reasoning about memory use of lazy functional programs is notoriously hard, so any help that the compiler can provide is extremely valuable. Unfortunately, heuristic approaches interacting with the many optimizations that exist already can lead to extreme difficulty in reasoning about memory consumption of a source program. We hope that in combination with future language tools, one could use the simplicity of the compiler presented here to better reason about time and space efficiencies, and help ensure that the compiler makes better decisions about how to preserve and/or improve time and space requirements. The lack of many intermediate representations makes this compiler an ideal target for such reasoning.

6.3 Theses Review

There are two theses presented in this dissertation. First, shared-environments efficiently implement call-by-need semantics, as described by the $\mathscr{CE}$ machine. This thesis is described in detail in Chapter 4. The conclusion of this chapter is that there are clear efficiencies gained and efficiencies lost with the shared environment approach. In particular, the efficiency gained by reduced thunk creation overhead enabled more efficient lambda calculus evaluation.

The primary conclusion for this thesis is that this is a good default behavior, but one that should be optimized away. Consider the analogue of strictness analysis. The idea behind strictness analyses and transforms is that laziness is a good default, but one that should be replaced with eager evaluation for efficiency where possible. In the same way, lazy thunk creation is a good default, but one that should be replaced with an optimized flat environment closure where necessary for efficiency reasons. In terms of cheap to create vs. efficient to execute, we can think of the shared-environment closure as the most extremely cheap to create, at the cost of efficiency to execute, whereas a just-in-time compiled closure specialized on its values might be at the other extreme: expensive to create, but efficient to execute.

The second thesis of the dissertation is that the simplicity of the compiler that implements the $\mathscr{CE}$ machine lends itself to formal reasoning. This thesis is described in detail in Chapter 5. The evidence for this thesis is provided in the form of a verified compiler. The implementation of the first verified compiler of a call-by-need semantics provides compelling support for this thesis.

The conclusion for this thesis is therefore that yes, the simplicity of the compiler indeed lends itself to formal reasoning. That said, there are still many open questions. For example, many of the proofs are excruciatingly complex. It is therefore not at all clear that the structure of the compiler and proofs is *the best possible*. Indeed, we expect there is room for improvement, particularly in the implementation of the proofs.

6.4 Conclusion

This dissertation has presented a novel technique for implementing call-by-need semantics using shared environments, the $\mathscr{CE}$ machine, along with a pair of compilers that provide evidence for the primary thesis of this dissertation: the $\mathscr{CE}$ machine lends itself to high performance, easy to reason about compilers.

Given that these are desirable properties for a compiler to have, then we must conclude that the $\mathscr{CE}$ machine and the compilers it enables are valuable tools for implementing call-by-need. My hope is that if the compilers developed in this dissertation are not used directly in future compilers for lazy languages, the ideas underlying them will be.

Appendices

Appendix A

Native Code Compiler Implementation Details

In this appendix we discuss details of the native code compiler. We focus on implementation details not included in Chapter 4. This includes a complete definition of the language, as well as extensive examples of what writing programs in the language looks like. In part, we share these details due to a number of interesting properties that the implementation has. Some of those properties include

- No first class recursion and let bindings. Replaced by use of Y combinator.
- No algebraic data types: Scott encoded data types.
- World type: reasoning about IO in an untyped setting by passing world values.
- No libc dependence: system calls as an explicit language construct.
- Novel syntax: explicit parentheses, a merging of lambda calculus and Lisp.

A.1 Source Language

For a better grasp on what the $\mathscr{CE}$ native code compiler can do, this section defines the syntax and core language, and provides a number of example programs and libraries. It highlights the fact that the compiler can handle machine literals and primitive operations, including system calls and a basic system for monadic side effects.

One way to view this section is as the results of an experiment in how simple we can make a call-by-need source language, and still have the ability to write real programs.

A.1.1 Language Definition

Here we give the full language definition:

```
data Expr a b = Var b
              | App (Expr a b) (Expr a b)
              | Lam a (Expr a b)
              | Lit Literal
              | Op Op
              | World

type Literal = Int

data Op = Add
        | Sub
        | Mul
        | Div
        | Mod
        | Eq
        | Neq
        | Lt
        | Gt
        | Le
        | Ge
        | Write WordSize
        | Read WordSize
        | Call String Int
        | Syscall Int
```

The expressions include the three standard lambda calculus constructors, as well as literals `Lit`, which are standard 32 bit machine words, machine operations `Op`, and world values `World`, which are used for reasoning about IO.

A.1.2 Syntax

The source syntax for the compiler is also, to the best of my knowledge, unique. There are a number of non-standard design decisions worth mentioning. The first, and most significant one, is a non-standard use of parentheses. Instead of implicit applications, we use `()` parentheses to explicitly denote left-associative applications, and `[]` brackets to denote explicit right-associative applications. We use either the unicode character for lambda `λx.b` the forward slash `\x.b` to denote lambdas, where `x` is a variable and `b` is the body of the lambda. This choice of explicit applications simplifies the syntax in a way that prevents ambiguity and simplifies parsing. It is effectively standard notation but with explicit parenthesis, like Lisp, making it easier to parse. In addition, we add syntax for let bindings in the form of curly braces. Read left curly braces `{` as `let` and right curly brace `}` as `in`. The reason for this was to avoid any keywords in the language, and therefore we require no tokenizer. Finally, we support string and character literals in the standard way. Following Haskell, we also define a global binding scope, implementing the following structure (with the default compilation options):

```
{
<preludes>
<program source>
} (main Ω λv.λw.w)
```

This approach allows us to write programs, much like Haskell, with global bindings, including a requirement to bind the variable `main` as our outermost function. From a typed perspective, this results in an expression of type World, assuming the input of `Ω` (the input world value). The syntax is easy enough to parse that it's worth sharing the parser source directly as a kind of formal grammar:

```
lc :: Parser SExpr
lc =  Lam <$> ((char '\\' <|> char 'λ') *> word <* char '.') <^> lc
  <|> Lit <$> literal
  <|> Op  <$> op
  <|> Var <$> word
  <|> World <$ char 'Ω'
  <|> char '(' ^> (foldl1 App <$> many1 (notCode *> lc <* notCode)) <^
      char ')'
  <|> char '[' ^> (foldr1 App <$> many1 (notCode *> lc <* notCode)) <^
      char ']'
  <|> char '\'' *> charLit <* char '\''
  <|> char '\"'
```

```
    *> (foldr (\h t -> App (App cons h) t) nil <$> many charLit) <*
    char '\"'
<|> char '{' ^>
    (lets <$> many (notCode *> binding <* notCode) <^>
    (char '}' ^> lc))
    where lets = flip $ foldr ($)
          binding = mylet <$> word <^> (char '=' ^> lc)
          mylet var term body = App (Lam var body) term
```

Note that due to our avoidance of tokenization, we directly parse any non-numeral, parentheses, or white space string into a variable.

A.1.3 Prelude

An extremely useful tool for any language is a base set of functionality available in global namespace. We follow Haskell terminology and refer to this set of bound variables as a Prelude.

We start with a pure fragment of the prelude: there are no partial functions modulo termination and type-safety. Note we have wrapper functions for our machine primitive operations that force evaluation of arguments. We also make heavy use of right-associative applications, wrapped with square brackets `[]`. By using this right associative operator, along with the implementation of literals, applying themselves to a continuation, we attain a method of forcing evaluation before applying a primitive operation. This is in contrast to using an infix operator, e.g., the $ operator in Haskell. Note the _ prefix for primitive operations.

Also note that we define the Y-combinator at the start of the prelude; this is the source of all recursion. Whether or not this is sufficient for all call-by-need recursion seems to be a topic for disagreement, but we have chosen, wherever feasible, to remove constructs from the core language when an equivalent semantics can be achieved without it.

```
# Recursion!
Y = \g.(\x.[g x x] \x.[g x x])

# Boolean integer comparisons
= = \n.\m.\t.\f.[m n \n.\m._=]
!= = \n.\m.\t.\f.[m n \n.\m._\=]
>= = \n.\m.\t.\f.[m n \n.\m._>=]
<= = \n.\m.\t.\f.[m n \n.\m._<=]
< = \n.\m.\t.\f.[m n \n.\m._<]
> = \n.\m.\t.\f.[m n \n.\m._>]

# Arithmetic
+ = \n.\m.[m n \n.\m._+]
- = \n.\m.[m n \n.\m._-]
-' = (0 -)
* = \n.\m.[m n \n.\m._*]
/ = \n.\m.[m n \n.\m._/]
% = \n.\m.[m n \n.\m._%]
^ = \n.(Y \^.\l.\n*.\m.(m = 0 n* (^ (n * n*) (m - 1))) n 1)

# Booleans
false = \t.\f.f
true = \t.\f.t
and = \a.\b.(a b false)
```

```
or = \a.\b.(a true b)
not = \a.(a false true)

# Maybe
Nothing = \n.\j.n
Just = \a.\n.\j.(j a)

# Either
Left = \v.\l.\r.(l v)
Right = \v.\l.\r.(r v)

# Lists
cons = \h.\t.\n.\c.(c h t)
: = cons
nil = true
; = nil
null = \l.(l true \h.\t.false)

# Tuple
pair = \f.\s.\p.(p f s)
fst = \p.(p \x.\y.x)
snd = \p.(p \x.\y.y)
uncurry = \f.\p.(p f)

# Monad
>>= = \c.\f.\w.(c w f)
>> = \c.\f.\w.(c w \v.f)
return = pair
```

```
when = \b.\a.(b a (return 0))


# Misc

gcd = (Y \gcd.\a.\b.([b 0 =] a (gcd b [a b %])))

lcm = \a.\b.(a * b / (gcd a b))

even = \n.(n % 2 = 0)

odd = \n.(n % 2 = 1)

id = \x.x

const = true

compose = \f.\g.\x.[f g x]

flip = \f.\b.\a.(f a b)

until = (Y \until.\p.\f.\x.(p x x (until p f (f x))))

map = (Y \map.\f.\xs.(xs nil \h.\t.(cons (f h) (map f t))))

length = \xs.(Y \length.\xs.\acc.(xs acc \h.\t.(length t (acc + 1))) xs 0)

foldl = (Y \foldl.\f.\a.\bs.(bs a \b.\bs.(foldl f (f a b) bs)))

foldr = (Y \foldr.\f.\a.\bs.(bs a \b.\bs.(f b (foldr f a bs))))

append = \as.\bs.(foldr cons bs as)

++ = append

reverse = (foldl (flip cons) nil)

any = \p.\xs.(foldr or false (map p xs))

all = \p.\xs.(foldr and true (map p xs))

concat = (foldr append nil)

concatMap = \f.(foldr (compose append f) nil)

filter = \c.(Y \filter.\ls.(ls nil \h.\t.(c h (cons h) id (filter t))))

scanl = (Y \scanl.\f.\q.\ls.(cons q (ls nil \x.\xs.(scanl f (f q x) xs))))

scanr = (Y \scanr.\f.\q.\ls.(ls (cons q nil)

  \x.\xs.(scanr f x q nil \q.\qs.(cons (f x q)))))

iterate = (Y \iterate.\f.\x.(cons x (iterate f (f x))))
```

```
repeat = (iterate id)

take = (Y \take.\n.\xs.(xs ; \x.\xs.(n <= 0 ; (: x (take (n - 1) xs)))))

replicate = \n.\x.(take n (repeat x))

cycle = \xs.[concat repeat xs]

drop = (Y \drop.\n.\xs.(xs ; \x.\xs.(n = 1 xs (drop (n - 1) xs))))

tail = (drop 1)

splitAt = \n.\xs.(pair (take n xs) (drop n xs))

takeWhile = (Y \takeWhile.\f.\xs.(xs nil

  \x.\xs.(f x (cons x (takeWhile f xs)) nil)))

dropWhile = (Y \dropWhile.\f.\xs.(xs nil

  \x.\xs.(f x (dropWhile f xs) (cons x xs))))

splitHalf = (Y \sh.\xs.(xs (pair ; ;)

  \h1.\t1.(t1 (pair (: h1 ;) ;) \h2.\t2.

  {tails = (sh t2)} (tails \f.\s.(

  (pair (: h1 f) (: h2 s)))))))

sort = \f.{merge = (Y \merge.\xs.\ys.(xs ys \x.\xts.(ys xs \y.\yts.(f x y

             (: x (merge xts ys)) (: y (merge xs yts))))))}

  (Y \ms.\l.(length l < 2 l (splitHalf l \f.\s.(merge (ms f) (ms s)))))

split = \f.(Y \split.\xs.(xs (pair nil nil) \h.\t.(f h

  (pair nil t)

  (split t \acc.\rem.(pair (cons h acc) rem)))))

showInt = \i.{showPosInt = (compose reverse (Y \showPosInt.\i.(i = 0

    nil

    (cons (i % 10 + 48) (showPosInt (i / 10))))))} [

  (i = 0 "0")

  (i < 0 (cons '-' (showPosInt (0 - i))))

  (showPosInt i)

]
```

```
readInt = (compose (Y \readInt.\n.(n 0 \h.\t.(h - 48 + (readInt t * 10))))
  (compose reverse (takeWhile \h.(and (h >= 48) (h < 58)))))
showBool = \b.(b "true" "false")
forever = \c.(Y \forever.(>> c forever))
isSpace = \c.(any (= c) " \n\t\r")
partition = \f.(Y \partition.\s.\acc.(s (pair acc nil)
  \h.\t.(f h (pair acc s)
  (partition t (append acc)))))
splitWhen = \f.(Y \splitWhen.\xs.(split f xs \l.\r.(r
  (l ; \h.\t.(: l ;))
  \h.\t.{cont = (splitWhen r)}(l cont \h.\t.(: l cont)))))
words = (splitWhen isSpace)
lines = (splitWhen (= '\n'))
splitArgs = (map (map fst) (splitWhen \p.(p \e.\c.(and (not e) (c = ' ')))
  (drop 1 (scanl \q.\x.(q \p.\c.(c = '\\' (pair true x) (pair false x)))
    (pair false ' ')))))
intersperse = \v.(Y \intersperse.\l.(l l \h.\t.(t l
  \h'.\t'.[(: h) (: v) intersperse t])))
unwords = (compose concat (intersperse " "))
if = id
from = (iterate (+ 1))
range = \s.\e.(take (e - s + 1) (from s))
then = \cont.\result.cont
sum = (foldr + 0)
max = \x.\y.(x > y x y)
min = \x.\y.(x < y x y)
unlines = (concatMap (flip append "\n"))
apply = (Y \al.\f.\l.(l f \h.\t.(al (f h) t)))
```

```
signum = \n.(n = 0 0 (n > 0 1 (-' 1)))

abs = \n.(n < 0 (0 - n) n)

zipWith = \f.(Y \zipWith.\as.\bs.(as nil \a.\at.(bs nil \b.\bt.
  (: (f a b) (zipWith at bt)))))

zip = (zipWith pair)

for = \as.\f.(Y \for.\as.(as (return id) \a.\as.(>> (f a) (for as))) as)

mapm_ = (flip for)

mapm = \f.\as.(Y \mapm.\as.\acc.(as (return acc)
  \a.\as.(>>= (f a) \c.(mapm as (cons c acc)))) as nil)

replace = \l.\v.(map (const v) l)

zip-with-default = \f.\d.(Y \zwd.\as.\bs.(as
  (replace bs d)
  \a.\at.(bs
    (replace as d)
    \b.\bt.(: (f a b) (zwd at bt)))))

strcmp = \s1.\s2.(all id (zip-with-default = false s1 s2))

lookup = \elem.\eqfun.(Y \lookup.\l.(l
  Nothing
  \h.\t.(h \key.\value.(eqfun elem key
    (Just value)
    (lookup t)))))

elem = \val.\func.\list.(lookup val func list false true)
```

A.1.4 Side Effects and Partial Functions

A necessary feature of a real-world programming language is the ability to have side effects. Here we define a small set of basic tools for implementing basic side effects, including memory reads and writes, and system calls. Note we take an approach that avoids depending on `libc`, implementing analogous functionality directly in lambda calculus with access to system calls.

For modeling side effects in lambda calculus, we take an approach inspired by the IO monad in Haskell. We extend our language with an object of type real world, and have our side-effectful functions consume and generate a new real-world value. This allows us to implement and use the monad bind and return functions directly. While we don't have infix operators or Haskell's do-notation, writing imperative programs is still entirely possible, though certainly more cumbersome, much like Wadler's early work with monads [46] but without infix operators.

```
## Memory read and write wrappers ##
mvq = \val.\addr.\w.[w addr val \a.\b.\w._@q]
rdq = \addr.\w.[w addr \a.\w._$q]
mvl = \val.\addr.\w.[w addr val \a.\b.\w._@l]
rdl = \addr.\w.[w addr \a.\w._$l]
mvs = \val.\addr.\w.[w addr val \a.\b.\w._@s]
rds = \addr.\w.[w addr \a.\w._$s]
mvb = \val.\addr.\w.[w addr val \a.\b.\w._@b]
rdb = \addr.\w.[w addr \a.\w._$b]
mv = mvq
rd = rdq

## System calls ##
sys_mmap = \addr.\len.\prot.\flags.\fd.\offset.\w.
  [w 9 offset fd flags prot len addr \a.\b.\c.\d.\e.\f.\t.\w._!6]
sys_write = \fd.\buf.\count.\w.[w 1 count buf fd \a.\b.\c.\t.\w._!3]
sys_read = \fd.\buf.\count.\w.[w 0 count buf fd \a.\b.\c.\t.\w._!3]
sys_open = \fname.\flags.\mode.\w.
```

```
  [w 2 mode flags fname \a.\b.\c.\t.\w._!3]
sys_close = \fd.\w.[w 3 fd \a.\t.\w._!1]
sys_exit = \code.\w.[w 60 code\a.\t.\w._!1]
sys_socket = \dom.\type.\prot.\w.[w 41 prot type dom \a.\b.\c.\t.\w._!3]
sys_getpid = \w.[w 39 \t.\w._!0]
sys_fork = \world.[world 57 \t.\w._!0]
sys_nanosleep = \rqtp.\rmtp.\w.[w 35 rmtp rqtp \a.\b.\t.\w._!2]
sys_munmap = \addr.\len.\w.[w 11 len addr \a.\b.\t.\w._!2]
sys_wait = \pid.\w.[w 61 0 0 pid \pid.\a.\b.\t.\w._!3]
sys_execve = \fname.\argv.\envp.\w.
  [w 59 envp argv fname \a.\b.\c.\t.\w._!3]
sys_getdents = \fd.\dirent*.\count.\w.
  [w 78 count dirent* fd \a.\b.\c.\t.\w._!3]

## Basic IO ##
pageSize = 4096
malloc = \len.(sys_mmap 0 len 255 34 (-' 1) 0)
free = \ptr.(sys_munmap ptr 1)
sleep = \sec.\nsec.(>>= (malloc 2) \p.
  (>> (mvq sec p)
  (>> (mvq nsec (p + 8))
  (>>= (sys_nanosleep p p) \r.
  (free p)))))
newPage = (malloc pageSize)
read-utf8-char-from-buf = \p.(>>= (rdb p) \c1.
  (c1 / 128 = 0 (return (pair c1  1))
    (>>= (rdb (p + 1)) \c2.{c2' = (2 ^ 8 * c2 + c1)}
  (c2 / 128 = 0 (return (pair c2' 2))
    (>>= (rdb (p + 2)) \c3.{c3' = (2 ^ 16 * c3 + c2')}
  (c3 / 128 = 0 (return (pair c3' 3)) (>>= (rdb (p + 3)) \c4.
  (return (pair (2 ^ 24 * c4 + c3') 4)))))))))
write-utf8-char-to-buf = \c.\p.[
  (c <= 0x7f (>> (mvb c p) (return 1)))
  (c <= 0x7ff (>> (mvb (c / 0x40 + 0xc0) p)
              (>> (mvb (c % 0x40 + 0x80) (p + 1)) (return 2))))
  (c <= 0xffff (>> (mvb (c / 0x1000 + 0xe0) p)
               (>> (mvb (c % 0x1000 / 0x40 + 0x80) (p + 1))
               (>> (mvb (c % 0x40 + 0x80) (p + 2))
               (return 3)))))
  (>> (mvb '?' p) (return 1))]
readStrBuf = (Y \rstr.\loc.\n.(n = 0
  (return nil)
  (>>= (read-utf8-char-from-buf loc) \p.(p \c.\n'.
  (>>= (rstr (loc + n') (n - n')) \s.(return (: c s)))))))
readStrBuf' = (Y \rstr.\loc.
  (>>= (read-utf8-char-from-buf loc) \p.(p \c.\n'.
  (c = 0
  (return nil)
  (>>= (rstr (loc + n')) \s.(return (: c s)))))))
putStrBuf = \size.\buf.{
  putStrBuf = (Y \putStrBuf.\loc.\s.
  {finished = (return (pair loc s))}
  (s
    finished
    \h.\t.(loc - buf >= (size - 4)
      finished
      (>>= (write-utf8-char-to-buf h loc) \n.(putStrBuf (loc + n) t)))))
  }
  (putStrBuf buf)
open = \fname.(>>= newPage \nameBuf.(
  >> (putStrBuf pageSize nameBuf fname)
  (sys_open nameBuf 66 0)))
openrd = \fname.(>>= newPage \nameBuf.(
  >> (putStrBuf pageSize nameBuf fname)
  (sys_open nameBuf 0 0)))
putPtrBuf = (Y \putPtrBuf.\loc.\s.(s
  (>> (mv 0 loc) (return (loc + 8)))
  \h.\t.(>> (mv h loc) (putPtrBuf (loc + 8) t))))
readPtrBuf = (Y \rstr.\loc.\n.(n <= 0
  (return nil)
  (>>= (rd loc) \p.(p = 0
    (return nil)
    (>>= (rstr (loc + 8) (n - 1)) \s.(return (: p s)))))))
writeFd = \fd.\str.(>>= newPage \iobuf.(>> (Y \writeFd.\str.
  (>>= (putStrBuf pageSize iobuf str) \p.(p \p.\s.
  (>>= (sys_write fd iobuf (p - iobuf)) \n.
  (nil? s (return n) (writeFd s))))
) str) (free iobuf)))
writeFile = \fname.\str.(newPage >>= \iobuf.(
  >>= (open fname) \fd.(
  >> (fd > 1024 (sys_exit 2) (writeFd fd str))
  (free iobuf))))
readFd = \fd.(>>= newPage \iobuf.(Y \readFd.
  (>>= (sys_read fd iobuf pageSize) \n.
  (>>= (readStrBuf iobuf n) \s.
  (n < pageSize
    (return s)
    (nil? s
      (return nil)
      (>>= readFd (compose return (append s)))))))))
readFile = \fname.(>>= (openrd fname) \fd.
  (or (fd > 1024) (fd < 0)
    (sys_exit (-' 1))
    (readFd fd)))
putStr = (writeFd 1)
putStrLn = \s.(writeFd 1 (append s "\n"))
print = putStrLn
getContents = (readFd 0)
getLine = (>>= getContents \s.(return (takeWhile (!= '\n') s)))
interact = \f.(>>= getContents (compose putStr f))

# Prelude-like utilities that require IO, e.g. partial functions
fork = \c.(>>= sys_fork \n.(n = 0 (>> c (sys_exit 0)) (sys_wait n)))
error = \s.(>> (putStrLn s) (sys_exit 1) 0)
undefined = (error "undefined")
PME = (error "Pattern match error")
head = \l.(l (error "head of empty list") \h.\t.h)
index = \n.\k.(Y \index.\n.\k.\cont.(n = 0
  cont
  \x.(index (n - 1) k (k = n x cont))) n (n - k + 1)
     (error "index failed"))
listIndex = (Y \listIndex.\n.\l.(l (error "listIndex failed")
  \h.\t.(n = 0 h (listIndex (n - 1) t))))
!! = listIndex
fst = (index 2 1)
snd = (index 2 2)
sprintf_ = \k.(Y \printf.\acc.\l.(l (k acc) \h.\t.(h = '%'
  (t (append acc "%") \h.\t'.[
    (h = 'd' \i.(printf (append acc (showInt i)) t'))
    (h = 'c' \c.(printf (append acc (cons c nil)) t'))
    (h = 's' \s.(printf (append acc s) t'))
    (h = 'b' \b.(printf (append acc (showBool b)) t'))
    (printf (append acc "%") t)])
  (printf (append acc (cons h nil)) t))) nil)
sprintf = (sprintf_ id)
printf = (sprintf_ print)
trace = \x.(sprintf_ print 0 \w.\s.x)
putWords = (Y \putWords.\ws.\startloc.(ws
  (return (pair startloc nil))
  \w.\ws.
    (>>= (putStrBuf pageSize startloc (++ w (: 0 ;))) \p.(p \endloc.\str.
    (>>= (putWords ws endloc) \p.(p \ptrloc.\ws.
    (return (pair ptrloc (cons startloc ws)))))))))
getArgs = (>>= sys_getpid \i.{fname = (sprintf "/proc/%d/cmdline" i)}
  (>>= (readFile fname) \s.[return tail [splitWhen = 0] s]))
getEnv = (>>= sys_getpid \i.{fname = (sprintf "/proc/%d/environ" i)}
  (>>= (readFile fname) \s.{bindings = (splitWhen (= 0) s)}
  (return (map (split (= '=')) bindings))))
getRawEnv = (>>= sys_getpid \i.{fname = (sprintf "/proc/%d/environ" i)}
  (>>= (readFile fname) \s.{bindings = (splitWhen (= 0) s)}
  (return bindings)))

# LS

d_ino = \ld.(rdl ld)

d_off = \ld.(rdl (ld + 8))

d_reclen = \ld.(rds (ld + 16))

d_name = \ld.(ld + 18)


# String -> IO [String]

ls = \path.(>>= newPage \buf.

  (>> (putStrBuf pageSize buf path)

  (>>= (sys_open buf 0x10000 0) \fd.

  (fd < 0

    (>> (printf "%s is not a directory" path)

    (return nil))

  (>>= (sys_getdents fd buf pageSize) \n.

  (n < 0

    (>> (printf "getdents on dir \"%s\" failed with errno %d" path n)

    (return nil))

  {read-dent = (Y \rdd.\ld.(ld - n >= buf

      (return nil)

    (>>= (d_ino ld) \ino.

    (>>= (d_off ld) \off.

    (>>= (d_reclen ld) \reclen.

    (>>= (readStrBuf' (d_name ld)) \name.

    (>>= (rdd (ld + reclen)) \names.

    (return (cons name names)))))))))

  }

  (>>= (read-dent buf) \files.
  (>> (free buf)
  (return files))))))))
pwd = (>>= getEnv \env.(lookup "PWD" strcmp env (return "/") return))
findPath = \name.(>>= getEnv \env.
  {paths = (lookup "PATH" strcmp env
    (cons "/usr/bin" nil) (splitWhen (= ':')))}
  {find-file = (Y \ff.\paths.(paths
    (>> (printf "couldn\'t find %s" name) (return name))
      \p.\ps.(>> (printf "checking %s")
    (>>= (ls p) \fs.(elem name strcmp fs
    (>> (printf "found %s in %s" name p)
    (return (append p (cons '/' name))))
    (>> (printf "couldn\'t find %s in %s" name p) (ff ps)))))))}
  (find-file paths))
exec = \str.(splitArgs str (return 0) \fname.\argv.
  (>>= getEnv \env.
  (>> (printf "running %s with args: " fname)
  (>> (mapm_ putStr argv) (>> (print ".")
  (>>= (elem '/' = fname (return fname) (findPath fname)) \fullname.
  (>>= newPage \buf.
  (>>= (putStrBuf pageSize buf (++ fullname (: 0 ;))) \p.(p \l.\str.
  (>>= (putWords argv l) \p.(p \al.\alocs.
  (>>= (putPtrBuf al (cons buf alocs)) \l.
  (>>= getRawEnv \env.
  (>>= (putWords env l) \p.(p \el.\elocs.
  (>>= (putPtrBuf el elocs) \l.
  (>>= (sys_execve buf al el) \retval.
  (>> (free buf)
  (return retval)))))))))))))))))
system = (compose fork exec)
```

Note the use of parentheses gets pretty hairy. Despite our right associative applications, the lack of infix operators and do-notation hurts the readability of long imperative programs significantly. Still, it is certainly *possible* to write programs in this style. Note also that our heavy use of newPage shows that our lack of stack allocated memory is a pain. Due to our use of the system stack for our argument closures and update markers, incorporating alloca-style stack allocations would be a significant challenge, though it should be possible.

A.1.5 Example Programs

In addition to re-writing the no-fib benchmark suite, we started writing example programs in this source language to gauge the difficulty. These programs use the prelude described above to implement basic programs.

We start with a canonical example of using call-by-need to implement a dynamic programming linear time Fibonacci.

```
fibs = (Y \fibs.[(1 :) (1 :) (zipWith + fibs (tail fibs))])
```

This is an interesting example, because unlike the similar Haskell implementation, this implementation is *guaranteed* to run in linear time. Because Haskell is a *non-strict* language, we rely on its use of call-by-need as an optimization technique, not a guaranteed property of the language. Note also the infix use of `:`. This is possible due to the implementation of machine integers. Recall that we don't have algebraic data types, so the technique used by Haskell of wrapping machine literals in a `newtype` is not an option. Instead, we use evaluation to force the value. In place of an environment pointer, we place the machine literal, and use the code pointer to take the continuation and apply itself to it. In this case, that results in the cons constructor applying itself to the value 1. Note that this is the approach used in all the machine primitive operations in the prelude listed above as well. While this is operationally equivalent the Haskell approach, it does raise questions about referential transparency.

Another simple example shows how we can use the fork system call to implement parallel programs. This shows how easy it is to use the IO monad without the use of type classes.

```
main = (>>= sys_fork \r.(r = 0
  (print "Child says hi")
  (print "Parent says hi")))
```

Note again we use the fact that forcing `r` before checking equality is a perfectly valid thing to do, though not necessary in this case as the equality check `=` will force it as well.

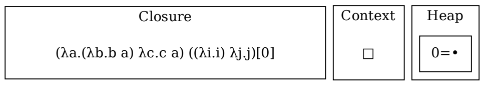
A.2 Visualization

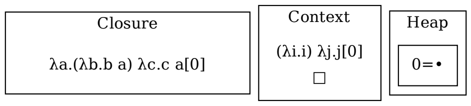
In addition to the native code compiler, I have implemented a graphviz-based visualization tool that helps visualize how the heap changes over time. It uses variable names in the heap instead of de Bruijn indices to increase readability. It also shows the stack and current closure through steps of the small-step $\mathscr{CE}$ semantics. The primary use is just to see how the cactus structure evolves through evaluation, and how it ensures sharing.

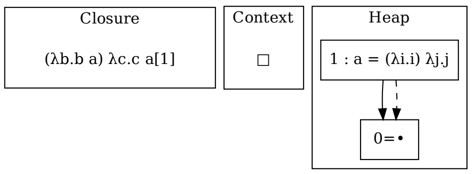
For example, consider evaluation of the expression

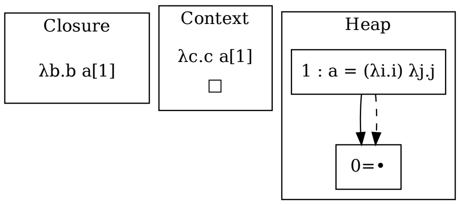
```
(\a.(\b.(b a) \c.(c a)) (\i.i \j.j))
```

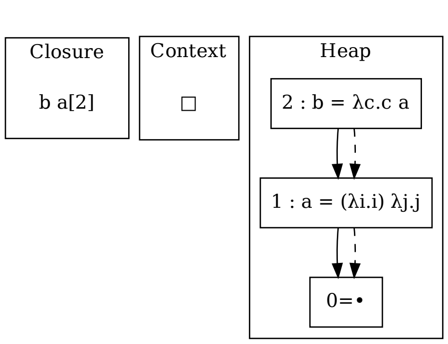
By running the `cem` executable with the flags `-pg`, we get the following output:

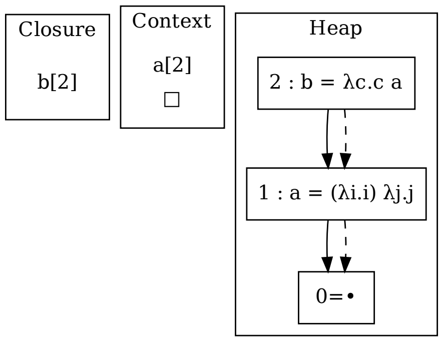


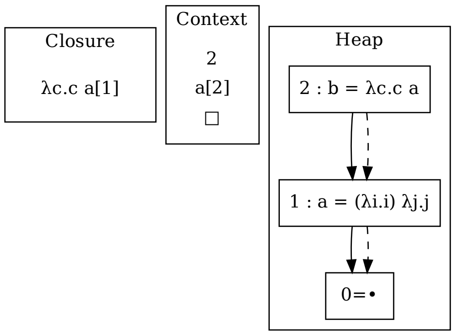


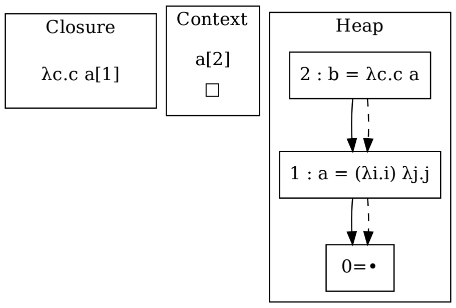


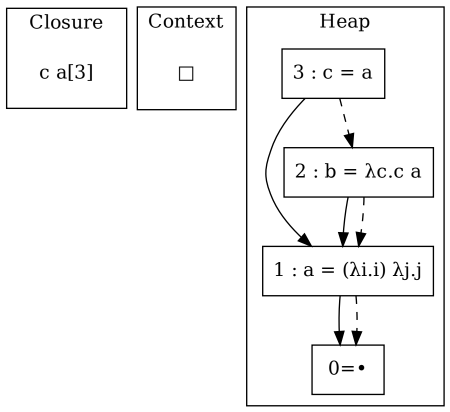


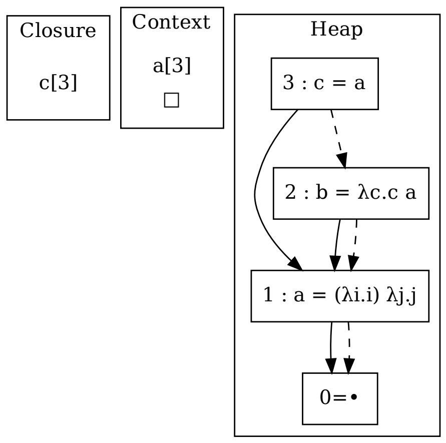


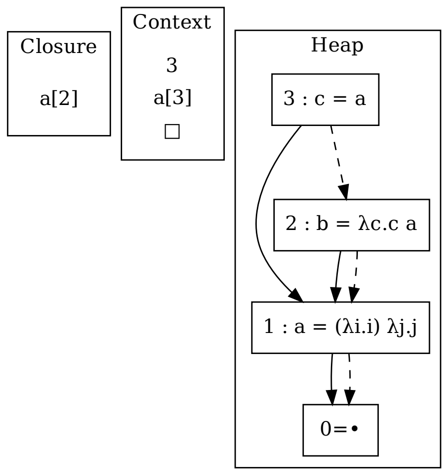


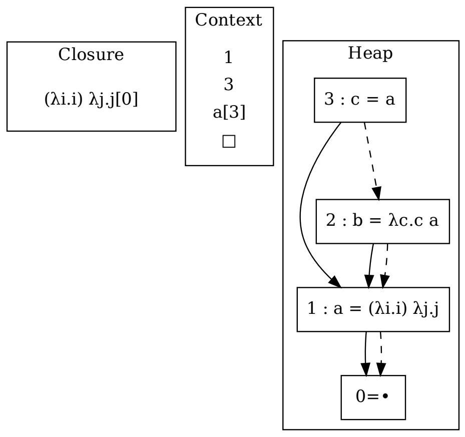


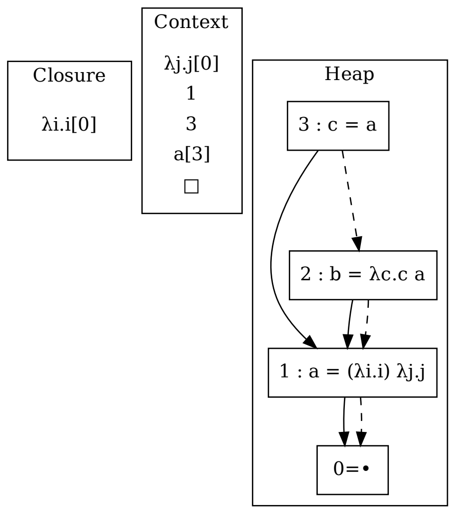


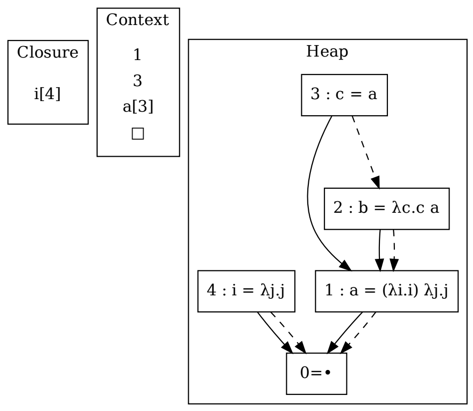

Closure
c a[3]
Context
□
Heap
3 : c = a
2 : b = λc.c a
1 : a = (λi.i) λj.j
0=•

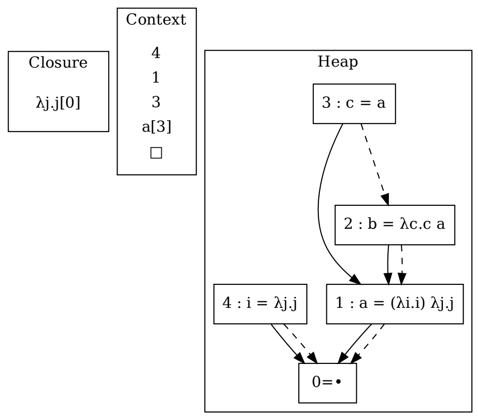

Closure
c[3]
Context
a[3]
□
Heap
3 : c = a
2 : b = λc.c a
1 : a = (λi.i) λj.j
0=•

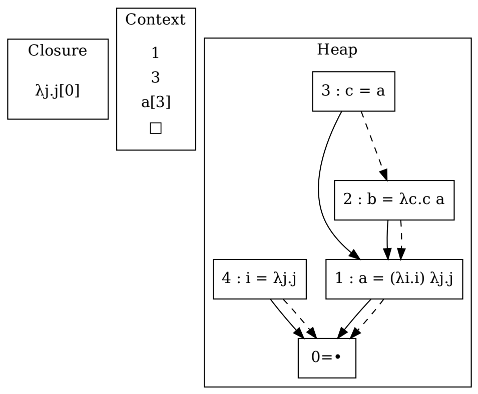

Closure
a[2]
Context
3
a[3]
□
Heap
3 : c = a
2 : b = λc.c a
1 : a = (λi.i) λj.j
0=•

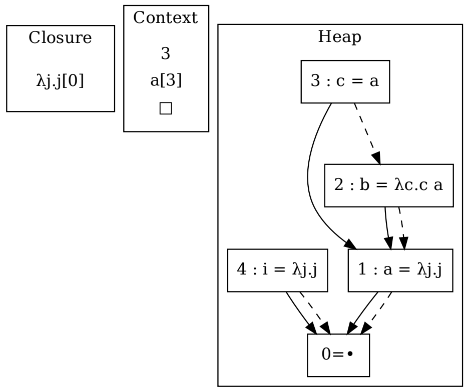
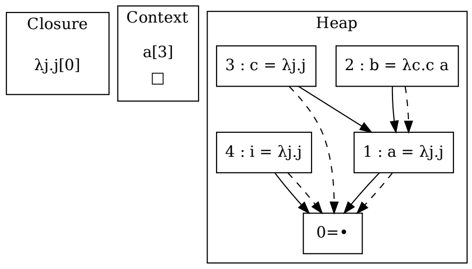
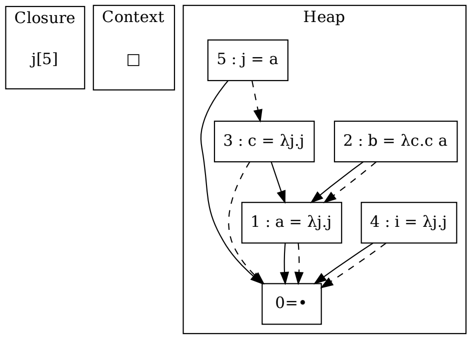
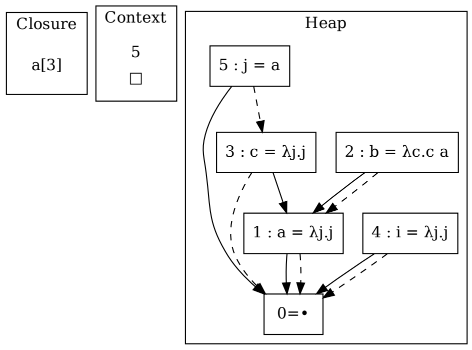
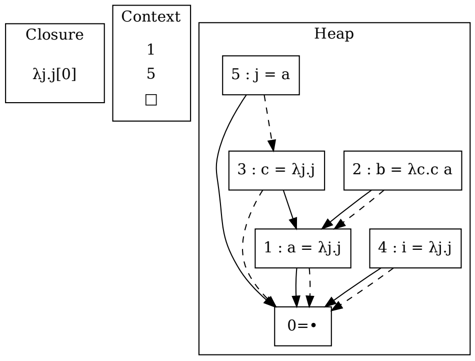
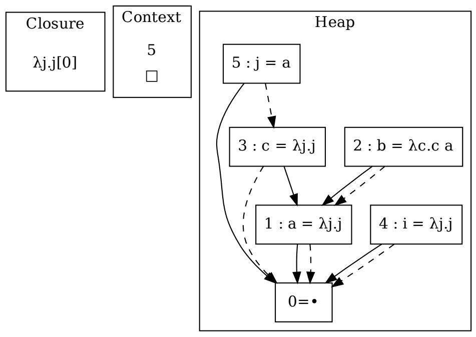

Closure
(λi.i) λj.j[0]
Context
1
3
a[3]
□
Heap
3 : c = a
2 : b = λc.c a
1 : a = (λi.i) λj.j
0=•

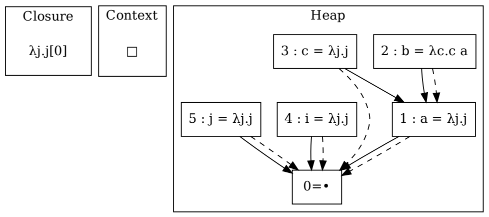

Closure
λi.i[0]
Context
λj.j[0]
1
3
a[3]
□
Heap
3 : c = a
2 : b = λc.c a
1 : a = (λi.i) λj.j
0=•


Closure
i[4]
Context
1
3
a[3]
□
Heap
3 : c = a
2 : b = λc.c a
4 : i = λj.j
1 : a = (λi.i) λj.j
0=•


Closure
λj.j[0]
Context
4
1
3
a[3]
□
Heap
3 : c = a
2 : b = λc.c a
4 : i = λj.j
1 : a = (λi.i) λj.j
0=•


Closure
λj.j[0]
Context
1
3
a[3]
□
Heap
3 : c = a
2 : b = λc.c a
4 : i = λj.j
1 : a = (λi.i) λj.j
0=•


Closure
λj.j[0]
Context
3
a[3]
□
Heap
3 : c = a
2 : b = λc.c a
4 : i = λj.j
1 : a = λj.j
0=•

Closure
λj.j[0]
Context
a[3]
□
Heap
3 : c = λj.j
2 : b = λc.c a
4 : i = λj.j
1 : a = λj.j
0=•

Closure
j[5]
Context
□
Heap
5 : j = a
3 : c = λj.j
2 : b = λc.c a
1 : a = λj.j
4 : i = λj.j
0=•

Closure
a[3]
Context
5
□
Heap
5 : j = a
3 : c = λj.j
2 : b = λc.c a
1 : a = λj.j
4 : i = λj.j
0=•

Closure
λj.j[0]
Context
1
5
□
Heap
5 : j = a
3 : c = λj.j
2 : b = λc.c a
1 : a = λj.j
4 : i = λj.j
0=•

Closure
λj.j[0]
Context
5
□
Heap
5 : j = a
3 : c = λj.j
2 : b = λc.c a
1 : a = λj.j
4 : i = λj.j
0=•



We can see that through the evaluation, the variable `a` is dereferenced twice, in two different scopes. The cactus structure ensures that the value is correctly shared between the two instances, with the second dereference correctly dereferencing the evaluated identity function.

A.2.1 Benchmarks

To convince the reader that the benchmarks are equivalent to the Haskell variants described in Section 4.2, we share the source code of some of them here, alongside their Haskell counterparts.

First, we start with the primes example. In our language, the implementation looks as follows.

```
divs = \n.\x.(x % n != 0)
thefilter = \l.(filter [divs head l] (tail l))
primes = [(map head) (iterate thefilter) from 2]

main = (>>= getArgs \args.(args (error "usage: primes <n>") \n.\t.
      (printf "%d" (!! (readInt n) primes))))
```

The Haskell version is as follows:

```
import System.Environment

suCC :: Int -> Int
suCC x = x + 1

isdivs :: Int  -> Int -> Bool
isdivs n x = mod x n /= 0

the_filter :: [Int] -> [Int]
the_filter (n:ns) = filter (isdivs n) ns

primes :: [Int]
primes = map head (iterate the_filter (iterate suCC 2))

main = do
[arg] <- getArgs
print $ primes !! (read arg)
```

Note that we use the helper function `from` from the prelude which uses the partially applied `+ 1` directly, but otherwise the implementations are identical. This is characteristic of how each of the benchmarks differ in our version compared to the Haskell version. Both the Haskell and our versions can be found in the source tarball.

Appendix B

Coq Implementation Details

Proof irrelevance is an important idea in philosophy of logic. It is the notion that we don't really need to care how the proof was built, only that it was sound. In machine checked proofs this notion is particularly relevant: if the machine checks the proofs of our lemmas, we can be sure that they are valid proofs.

In contrast, we *must* care about the contents of the definitions and theorems. Without them, the reader can't be sure what's been proved is what is being claimed to have been proved. Therefore, we use this section to define and discuss select Coq definitions and theorem statements in the implementation and proof of correctness of the verified compiler. We attempt to convey what worked well, and what posed significant challenges in the hope of informing future work in this area. The formalization is available at `https://github.com/stelleg/cem_coq` and has been successfully type-checked with Coq version 8.9. The version used for this dissertation is tagged with `dissertation`, which at the time of writing enables it to be downloaded as a GitHub release tarball. Inevitably, we will use definitions from both the Coq standard libraries, or generic definitions implemented throughout the project. This appendix covers only the most imporant definitions and theorems.

It's also worth noting that the reader will find many definitions, lemmas, algorithms, and proofs in the repository that are unused in the final proof. This is a byproduct of the process of proving in a computer logic. Often, when one starts a proof it seems like a certain property or lemma will be required, so one proves that lemma, only to discover later that it was not necessary. Such helper lemmas sometimes prove useful later, during a different proof. Hopefully some of the unused code will prove useful in future work.

Another point worth noting is that the Coq code makes heavy use of Unicode characters and notations. In hindsight, the heavy use of fancy notations was almost always a mistake, and a refactoring to remove them would likely improve readability. Limiting use of notations to the occasional infix operator is likely a good middle ground.

B.1 Big-Step $\mathscr{CE}$ Semantics

Here we define the big-step syntax and semantics, which we use as our input language semantics. This is a formalization of the Figure 3.2.

We start with the lambda calculus with de Bruijn indices.

```
Inductive tm : Type :=
  | var : nat → tm
  | lam : tm → tm
  | app : tm → tm → tm.
```

Next, we define other helper definitions, including closures, environments (`env`), cells, heaps, configurations, what doing a lookup into the shared environment means (`clu`), and what replacing a closure at a location in a heap means (`update`).

```
Definition env := nat.

Record closure : Type := close {
  cl_tm : tm;
  cl_en : env
}.

Record cell : Type := cl {
  cell_cl : closure;
  cell_env : env
}.

Definition heap := Map nat cell.

Record configuration : Type := conf {
  conf_h : heap;
  conf_c : closure
}.

Fixpoint clu (v env:nat) (h:heap) : option (nat * cell) :=
  match lookup env h with
  | None => None
  | Some (cl c a) => match v with
    | S n => clu n a h
    | 0 => Some (env, cl c a)
    end
  end.
```

```
Fixpoint update (h : heap) (l : env) (v : closure) : heap := match h with
  | [] => []
  | (u, cl c e)::h => if beq_nat l u
    then (u, cl v e) :: update h l v
    else (u, cl c e) :: update h l v
  end.
```

Finally, we define the actual machine big-step semantics.

```
Reserved Notation " c1 '⇓' c2 " (at level 50).
Inductive step : configuration → configuration → Type :=
  | Id : ∀ M x y z Φ Ψ v e, clu y e Φ = Some (x, {M, z}) →
                   ⟨Φ⟩M ⇓ ⟨Ψ⟩v →
    ⟨Φ⟩close (var y) e ⇓ ⟨update Ψ x v⟩v
  | Abs : ∀ N Φ e, ⟨Φ⟩close (:λN) e ⇓ ⟨Φ⟩close (:λN) e
  | App : ∀ N M B B' Φ Ψ Y f e ne ae, isfresh (domain Ψ) f → f > 0 →
          ⟨Φ⟩close M e ⇓ ⟨Ψ⟩close (:λB) ne →
      ⟨Ψ, f ↦ {close N e, ne}⟩close B f ⇓ ⟨Y⟩close (:λB') ae   →
              ⟨Φ⟩close (M@N) e ⇓ ⟨Y⟩close (:λB') ae
where " c1 '⇓' c2 " := (step c1 c2).
```

B.2 Small-Step $\mathscr{CE}$ Semantics

Our small-step semantics is a straightforward implementation of the big-step semantics. The language is the same lambda calculus with de Bruijn indices, while we introduce a stack to match marker updates and argument bindings to variables. This is a formalization of Figure 3.3.

Most of the syntax is shared with the big-step semantics, and is imported directly from the big-step module. We share the definitions for stacks and states, followed directly by the small-step semantics.

```
Definition stack := list (closure + nat).

Inductive state : Type := st {
  st_hp : heap;
  st_st : stack;
  st_cl : closure
}.

Reserved Notation " c1 '→_s' c2 " (at level 50).
Inductive step : transition state :=
  | Upd : ∀ Φ b e l s,
  st Φ (inr l::s) (close (lam b) e) →_s
  st (update Φ l (close (lam b) e)) s (close (lam b) e)
  | Var : ∀ Φ s v l c e e', clu v e Φ = Some (l,cl c e') →
  st Φ s (close (var v) e) →_s st Φ (inr l::s) c
  | Abs : ∀ Φ b e f c s, isfresh (domain Φ) f → f > 0 →
  st Φ (inl c::s) (close (lam b) e) →_s st ((f, cl c e):: Φ) s (close b f)
  | App : ∀ Φ e s n m,
  st Φ s (close (app m n) e) →_s st Φ (inl (close n e)::s) (close m e)
where " c1 '→_s' c2 " := (step c1 c2).
```

B.2.1 Relation to Big-Step

We prove that the small-step semantics implements the big-step semantics with the following lemma. The description of the proof can be found in Section 5.2.1.

```
Notation " c1 '→_s*' c2 " :=
  (refl_trans_clos cesm.step c1 c2) (at level 30).

Lemma cem_cesm : ∀ Φ Ψ c v,
  conf Φ c ⇓ conf Ψ v → ∀ s,
  st Φ s c →_s* st Ψ s v.
```

B.3 Instruction Machine

Finally, we change representations and describe the assembly language of the abstract instruction machine; the target of the verified compiler. This is the formalization of language defined in Figure 5.1. Note the use of coercions to help ease the use of type-safe read and write operands, and an infix operator to duplicate common syntax for real world assembly languages when indexing an offset. Note that for our purposes, we only ever need a constant offset, but one could easily extend the language to allow for offsets to be defined by a read operand.

```
Definition Word := nat.
Definition Ptr := nat.

Inductive Reg :=
  | IP
  | EP
  | R1
  | R2.

Inductive WO :=
  | WR : Reg → WO
  | WM : Reg → nat → WO.

Coercion WR : Reg >-> WO.
Infix "%" := WM (at level 30).

Inductive RO :=
  | RW : WO → RO
  | RC : nat → RO.

Coercion RW : WO >-> RO.
Coercion RC : nat >-> RO.

Inductive Instr : Type :=
  | push : RO → Instr
  | pop : WO → Instr
  | new : nat → WO → Instr
  | mov : RO → WO → Instr.

Inductive BasicBlock : Type :=
  | instr : Instr → BasicBlock → BasicBlock
  | jump : option (RO*Ptr) → RO → BasicBlock.

Definition Program := list BasicBlock.
```

Finally, note that we use a list of basic blocks as our program. This list is indexed by integers in the machine semantics, so translating this to a Harvard architecture machine semantics should be relatively straightforward.

Now that we have the language for our abstract instruction machine target, we can look at the formal semantics for it. Without loss of generality, we choose a step relation on instructions, wrapped by a step instruction on full basic blocks. This eases reasoning about the relation to the small-step $\mathscr{CE}$ relation. This is a formalization of the semantics described in Figure 5.2.

We start with a register file and machine state, along with a straightforward semantics for reading and writing from operands given a machine state. We omit some uninteresting helper functions.

```
Inductive RegisterFile := mkrf {
  ip : Ptr;
  ep : Ptr;
  r1 : Ptr;
  r2 : Ptr
}.

Inductive State := st {
  st_rf : RegisterFile;
  st_p : Program;
  st_s : Stack;
  st_h  : Heap
}.

Open Scope nat_scope.
Inductive read : RO → State → Ptr → Type :=
  | read_reg : ∀ r s, read (RW (WR r)) s (rff (st_rf s) r)
  | read_mem : ∀ r o rf p s h v,
    lookup (o+rff rf r) h = Some v →
    read (RW (WM r o)) (st rf p s h) v
  | read_const : ∀ c s, read (RC c) s c.

Inductive write : WO → Word → State → State → Type :=
  | write_reg : ∀ r rf p s h w,
    write (WR r) w (st rf p s h) (st (upd r w rf) p s h)
  | write_mem : ∀ r o rf p s h w,
    write (WM r o) w (st rf p s h)
                     (st rf p s (replace beq_nat (o+rff rf r) w h)).
```

Given our machine definitions and read and write semantics, we can move directly to defining the step relation for instructions. The `step_bb` relation defines the execution of a full basic block by induction on the instructions contained in that basic blocks. The `step` relation then wraps that relation and defines how the state changes given execution of the basic block pointed to by the instruction pointer. Note that we don't define an explicit halting relation. Instead, the machine will be stuck when the current code to be executed is a value and the stack is empty, which is where the small-step $\mathscr{CE}$ semantics also get stuck. In contrast, the native code implementation has a check to ensure the stack is non-empty. A full compiler with a sufficiently sophisticated type system could likely do away with this check, only terminating with the `main` function returning a value of type `World`.

```
Inductive step_bb : BasicBlock → State → State → Type :=
  | step_push : ∀ rf p s h ro is v sn,
  read ro (st rf p s h) v →
  step_bb is (st rf p (v::s) h) sn →
  step_bb (instr (push ro) is) (st rf p s h) sn
  | step_pop : ∀ rf p s h wo is w s' sn,
  write wo w (st rf p s h) s' →
  step_bb is s' sn →
  step_bb (instr (pop wo) is) (st rf p (w::s) h) sn
  | step_new : ∀ rf p s h wo is w s' n sn,
  (∀ i, i < n → not (In (i+w) (domain h))) →
  w > 0 →
  write wo w (st rf p s (zeroes n w ++ h)) s' →
  step_bb is s' sn →
  step_bb (instr (new n wo) is) (st rf p s h) sn
  | step_mov : ∀ s is ro wo s' v sn,
  read ro s v → write wo v s s' →
  step_bb is s' sn →
  step_bb (instr (mov ro wo) is) s sn
  | step_jump0 : ∀ ro k j s s',
  read ro s 0 →
  write (WR IP) k s s' →
  step_bb (jump (Some (ro, k)) j) s s'
  | step_jumpS : ∀ ro k j s s' l k',
  l > 0 →
  read ro s l →
  read j s k →
  write (WR IP) k s s' →
  step_bb (jump (Some (ro, k')) j) s s'
  | step_jump : ∀ ro s s' l,
  read ro s l →
  write (WR IP) l s s' →
```

```
  step_bb (jump None ro) s s'
.

Inductive step : transition State :=
  | enter : ∀ rf p s h k bb sn,
    read IP (st rf p s h) k →
    nth_error p k = Some bb →
    step_bb bb (st rf p s h) sn →
    step (st rf p s h) sn.
```

With our step function defined, we can start thinking about how we want to implement our compiler, and how to relate the instruction machine to the small step semantics of our input language.

B.3.1 Relation to Small-Step Semantics

We now have everything we need, except for the compiler definition, which we will address in the next section. We start by assuming a `new` function, which is effectively a `malloc` that given a heap, returns a non-null free contiguous region of memory. Of course, in reality `malloc` is a partial function, but we aren't modelling running out of memory in this work.

```
Variable new : ∀ (n:nat) (h : im.Heap), sigT (λ w:nat,
  prod (∀ i, lt i n → (i+w) ∉ domain h)
       (w > 0)).

Definition prog_eq (p : Ptr) (pr : Program) (t : tm) :=
  let subpr := assemble t p in subpr = firstn (length subpr) (skipn p pr).
```

Next, we discuss heap relations between the instruction machine and the $\mathscr{CE}$ machine semantics. Unfortunately, this gets incredibly involved. Despite much effort, I was unable to find a more elegant relation between the two.

```
Inductive heap_rel : cesm.heap → im.Heap → Type :=
  | heap_nil : heap_rel [] []
  | heap_cons : ∀ l l' ne ch ih ip ep ine e t,
    l ∉ domain ch → l' ∉ domain ih →
    S l' ∉ domain ih → S (S l') ∉ domain ih →
    l > 0 → l' > 0 →
    heap_rel ch ih →
    heap_rel
      ((l, cl (close t e) ne)::ch)
      ((l', ip)::(S l', ep)::(S (S l'), ine)::ih).

Fixpoint in_heap_rel (ch : cesm.heap) (ih : im.Heap)
                     (r : heap_rel ch ih)
                     (l e ne : nat) (t : db.tm)
                     (il ip ep nep : Ptr) : Type := match r with
  | heap_nil => False
  | heap_cons l' il' ne' cht iht ip' ep' nep' e' t' _ _ _ _ _ _ rt =>
    if andb (beq_nat l l') (beq_nat il il') then
    (ne' = ne) *  (ip' = ip) * (ep' = ep) *
    (nep' = nep) * (e' = e) * (t' = t)
    else
      if andb (negb (beq_nat l l')) (negb (beq_nat il il'))
        then in_heap_rel cht iht rt l e ne t il ip ep nep
        else False
  end.

Inductive env_eq (ch : cesm.heap) (ih : im.Heap) (r : heap_rel ch ih)
  : nat → Ptr → Type :=
  | e0 : env_eq ch ih r 0 0
  | eS : ∀ l e ne t il ip ep nep,
    in_heap_rel ch ih r l e ne t il ip ep nep →
    env_eq ch ih r l il.

Inductive heap_eq (ch : cesm.heap) (ih : im.Heap)
                  (r : heap_rel ch ih) (p : Program) : Type :=
  | mkheap_eq :
    (∀ l e ne t il ip ep nep,
      in_heap_rel ch ih r l e ne t il ip ep nep →
      (prog_eq ip p t) *
      (env_eq ch ih r e ep) *
      (env_eq ch ih r ne nep)) →
    heap_eq ch ih r p.
```

In words, the heap_rel defines a heap relation that simply relates an entry in the $\mathscr{CE}$ heap to the three machine words in the instruction machine heap. in_heap_rel is a decidable relation that determines whether or not a given heap location and cell at that location and their corresponding machine analogues are related by a heap relation object. We say environments are equal if they are either both the empty environment or the first element of the environments are in the heap relation, and the tails are equal. Two heaps are equivalent if for every cell, the term portions are equivalent and the environment and environment continuations are equivalent. The inductive relation on the environments is crucial for inductive reasoning on de Bruijn indices.

Given our heap and environment relations, we can move to notions of closure, stack, and complete state equivalences.

```
Inductive clos_eq (ch : cem.heap) (ih : im.Heap)
                  (r : heap_rel ch ih) (p : Program):
                  closure → Ptr → Ptr → Type :=
  | c_eq : ∀ t e ip ep,
           prog_eq ip p t →
           env_eq ch ih r e ep →
           clos_eq ch ih r p (cem.close t e) ip ep.

Inductive stack_eq (ch : cem.heap) (ih : im.Heap)
                   (r : heap_rel ch ih) (p : Program) :
  cesm.stack → im.Stack → Type :=
  | stack_nil : stack_eq ch ih r p nil nil
  | stack_upd : ∀ l e ne t il ip ie ine cs is,
                in_heap_rel ch ih r l e ne t il ip ie ine →
                stack_eq ch ih r p cs is →
                stack_eq ch ih r p (inr l::cs) (0::il::is)
  | stack_arg : ∀ ip ep cs is c,
                 ip > 0 →
                 clos_eq ch ih r p c ip ep →
                 stack_eq ch ih r p cs is →
                 stack_eq ch ih r p (inl c::cs) (ip::ep::is).

Inductive state_rel (cs : cesm.state) (is : im.State) : Type :=
  | str : ∀ r,
  heap_eq (st_hp cs) (st_h is) r (st_p is) →
  clos_eq (st_hp cs) (st_h is) r (st_p is) (st_cl cs)
          (rff (st_rf is) IP) (rff (st_rf is) EP) →
  stack_eq (st_hp cs) (st_h is) r (st_p is) (st_st cs) (st_s is) →
  state_rel cs is.
```

For the stack relation, we have either the empty stacks or two stacks with either both update markers or both argument closures on top, and equivalent stacks below that. For update markers, we have that the two marker locations exist in the heap relation. For argument closures, we have that the closures are equivalent.

The closure and state relations are straightforward, the closure relation requires that the term and subprogram are equal, and that the environments are equivalent. The state relation requires equivalent closure and registerfile entries (IP and EP), equivalent heaps, and equivalent stacks.

Finally, we can pose our lemma that the instruction machine implements the small-step semantics.

```
Lemma cesm_im : ∀ v s s', state_rel s s' →
  cesm.step s v →
  sigT (λ v', prod (refl_trans_clos im.step s' v') (state_rel v v')).
```

This states that if we have related small-step and instruction machine states, and we take a small step, then the instruction machine will take a step to a state that is related to the new state of the small-step machine. While we don't discuss the proof in detail, it's worth mentioning that even with many supporting definitions and lemmas, it took on the order of 2000 Ltac (Coq's tactic language) statements to prove. Note that is a single step, but the reflexive transitive closure version follows trivially.

B.4 Compiler and Correctness

We've seen the `assemble` function referred to relate the lambda terms to the machine instructions in the previous section. This sections defines that function, which is the entire compiler implementation.

```
Infix ";" := instr (at level 30, right associativity).

Fixpoint var_inst (i : nat) : BasicBlock := match i with
  | 0 => push EP ;
         push (RC 0) ;
         mov (EP%0) R1 ;
         mov (EP%1) EP ;
         jump None R1
  | S i => mov (EP%2) EP ;
           var_inst i
  end.



Fixpoint assemble (t : tm) (k : nat) : Program := match t with
  | var v => [var_inst v]
  | app m n => let ms := assemble m (1+k) in
               let nk := 1+k+length ms in
                push EP ;
                push (RC nk) ;
                jump None (RC (1+k)) ::
                ms ++
                assemble n nk
  | lam b => pop R1 ;
             jump (Some (RW (WR R1), (1+k))) (RC (2+k)) ::
             (*Update*)
             pop R1 ;
             mov (RC k) (R1%0) ;
             mov EP (R1%1) ;
             jump None (RC k) ::
             (*Take*)
             new 3 R2 ;
             mov R1 (R2%0);
             pop (R2%1) ;
             mov EP (R2%2) ;
             mov R2 EP ;
             jump None (3+k) ::
             assemble b (3+k)
  end.
```

We can see that the compiler is very simple, only requiring 35 lines of code. It is this simplicity that enables the formal reasoning achieved in the previous section.

Finally, we can define our top level correctness theorem.

```
Definition compile t := assemble t 0.

Theorem compile_correct (t : db.tm) v : cem.step (cem.I t) v →
  sigT (λ v', refl_trans_clos im.step (im.I (compile t)) v' *
            state_rel (cesm.st (cem.conf_h v) nil (cem.conf_c v)) v').
```

This theorem states that if a term steps in the big-step semantics to a value, then the instruction machine will step in zero or more steps to a related state. Knowledge of the `state_rel` relation is crucial here: it would be trivial to define a meaningless relation, e.g. the relation defined by `λ c i, True`, and prove the relation trivially. Because we know that the relation requires equivalence of term and subprogram pointed to by IP, we know that the theorem is what we want.

B.5 Curien

To relate our implementation to a known semantics, we choose Curien's calculus of closures. We show that the call-by-name $\mathscr{CE}$ semantics implement Curien's calculus of closures. We start by defining Curien's calculus of closures.

```
Inductive closure := | close : tm → list closure → closure.
Definition env := list closure.

Inductive step : closure → closure → Type :=
  | Abs : ∀ b e, step (close (lam b) e) (close (lam b) e)
  | Var : ∀ x e v c,
      nth_error e x = Some c →
      step c v →
      step (close (var x) e) v
  | App : ∀ m n b e v mve,
      step (close m e) (close (lam b) mve) →
      step (close b (close n e::mve)) v →
      step (close (app m n) e) v.
```

This is a formalization of the semantics in Figure 2.1. We relate this to the call-by-name variant of the $\mathscr{CE}$ semantics. Defined in Figure 3.1, we formalize this semantics in Coq as follows.

```
Reserved Notation " c1 '⇓n' c2 " (at level 50).
Inductive step : configuration → configuration → Type :=
  | Id : ∀ M x z Φ Ψ y v e, clu y z Φ = Some (x, {M, e}) →
            ⟨Φ⟩M ⇓n ⟨Ψ⟩v →
    ⟨Φ⟩close (var y) z ⇓n ⟨Ψ⟩v
  | Abs : ∀ N Φ e, ⟨Φ⟩close (:λN) e ⇓n ⟨Φ⟩close (:λN) e
  | App : ∀ N M B B' Φ Ψ Y f e ne ae, isfresh (domain Ψ) f →
          ⟨Φ⟩close M e ⇓ ⟨Ψ⟩close (:λB) ne →
      ⟨Ψ, f ↦ {close N e, ne}⟩close B f ⇓ ⟨Y⟩close (:λB') ae   →
              ⟨Φ⟩close (M@N) e ⇓ ⟨Y⟩close (:λB') ae
where " c1 '⇓n' c2 " := (step c1 c2).
```

Note the only difference between this semantics and the call-by-need is the lack of an updated heap in the `Id` rule (and we don't require nonzero heap locations, but this is unimportant). With these two semantics defined, we can relate them by first relating environments and closures. We start by defining an inductive relation on what it means for environments to be equivalent. A cactus environment is equivalent a Curien environment if following the environment pointers results in equivalent closures at each location.

```
Inductive env_eq (h : heap) : curien.env → cem.env → Type :=
  | env_eq_nil : ∀ l, env_eq h nil l
  | env_eq_cons : ∀ t e e' l l' l'',
      lookup l h = Some (cl (cem.close t l') l'') →
      env_eq h e l' →
      env_eq h e' l'' →
      env_eq h (curien.close t e :: e') l
  .
Inductive close_eq (h : heap) : curien.closure → cem.closure → Type :=
  | close_equiv : ∀ t e l,
      env_eq h e l →
      close_eq h (curien.close t e) (cem.close t l).
```

With this relation, we can define and prove a helper lemma that takes advantage of the monotonicity of the cactus environment to implement a cactus environment to prove that if an environment is equivalent to a Curien environment, and the call-by-name $\mathscr{CE}$ semantics take a step, it will stay equivalent. This is noteworthy as one of the few places in the proof structures that would need to be changed to incorporate a notion of heap reuse/garbage collection.

```
Lemma env_eq_step : ∀ h h' c v e l,
  cem_name.step (conf h c) (conf h' v) →
  env_eq h e l →
  env_eq h' e l.
```

Finally, we are ready to define our correctness lemma.

```
Theorem step_eq : ∀ h c c' v, close_eq h c c' → curien.step c v →
  sigT (λ co, match co with conf h' v' => prod
    (cem_name.step (conf h c') co)
    (close_eq h' v v')
  end).
```

Note the use of `sigT` instead of standard `∃` syntax. This is due to our use of the `Type` universe for all of our judgements, instead of `Prop`. I do this to enable the eventual possibility of doing computation on the judgements, to allow reasoning about time and space requirements of the semantics. The impredicative nature of `Prop` was never required in the proofs of this dissertation, so the use of `Type` was sufficient.

References

[1] Andrew W Appel, *Compiling with Continuations*, Cambridge University Press, 1992.

[2] Andrew W Appel and Trevor Jim, *Optimizing closure environment representations*, Princeton University, Department of Computer Science, 1988.

[3] Andrew W Appel and David B MacQueen, *Standard ML of New Jersey*, Programming Language Implementation and Logic Programming, Springer, 1991, pp. 1–13.

[4] Zena M Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler, *A call-by-need lambda calculus*, Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, ACM, 1995, pp. 233–246.

[5] Hendrik Pieter Barendregt, *The Lambda Calculus*, vol. 3, North-Holland Amsterdam, 1984.

[6] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al., *The Coq proof assistant reference manual: Version 6.1*, Ph.D. thesis, Inria, 1997.

[7] Małgorzata Biernacka and Olivier Danvy, *A concrete framework for environment machines*, ACM Transactions on Computational Logic (TOCL) **9** (2007), no. 1, 6.

[8] Urban Boquist and Thomas Johnsson, *The GRIN project: A highly optimising back end for lazy functional languages*, Implementation of Functional Languages, Springer, 1997, pp. 58–84.

[9] Joachim Breitner, *Lazy evaluation: From natural semantics to a machine-checked compiler transformation*, Ph.D. thesis, Karlsruher Instituts für Technologie, 2017.

[10] Adam Chlipala, *A certified type-preserving compiler from lambda calculus to assembly language*, PLDI '07: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, vol. 42, ACM, 2007, pp. 54–65.

[11] P-L Curien, *An abstract framework for environment machines*, Theoretical Computer Science **82** (1991), no. 2, 389–402.

[12] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny, *On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation*, Theoretical Computer Science **435** (2012), 21–42.

[13] Olivier Danvy and Ian Zerny, *A synthetic operational account of call-by-need evaluation*, Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming, ACM, 2013, pp. 97–108.

[14] Stephan Diehl, Pieter Hartel, and Peter Sestoft, *Abstract machines for programming language implementation*, Future Generation Computer Systems **16** (2000), no. 7, 739–751.

[15] Atze Dijkstra, Jeroen Fokker, and S Doaitse Swierstra, *The architecture of the Utrecht Haskell compiler*, Proceedings of the 2nd ACM SIGPLAN symposium on Haskell, ACM, 2009, pp. 93–104.

[16] Jon Fairbairn and Stuart Wray, *TIM: A simple, lazy abstract machine to execute supercombinators*, Functional Programming Languages and Computer Architecture, Springer, 1987, pp. 34–45.

[17] Daniel Friedman, Abdulaziz Ghuloum, Jeremy Siek, and Onnie Winebarger, *Improving the lazy Krivine machine*, Higher-Order and Symbolic Computation **20** (2007), 271–293, 10.1007/s10990-007-9014-0.

[18] EA Hauck and Ben A Dent, *Burroughs' B6500/B7500 stack mechanism*, Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, ACM, 1968, pp. 245–251.

[19] J. Hughes, *Why functional programming matters*, The Computer Journal **32** (1989), no. 2, 98–107.

[20] Jean Ichbiah, *Rationale for the design of the Ada programming language*, Cambridge University Press, 1991.

[21] Peter Z Ingerman, *A way of compiling procedure statements with some comments on procedure declarations*, Commun. ACM **4** (1961), no. 1, 55–58.

[22] Thomas Johnsson, *Efficient compilation of lazy evaluation*, SIGPLAN Notices, 1984.

[23] J.L. Krivine, *A call-by-name lambda-calculus machine*, Higher-Order and Symbolic Computation **20** (2007), no. 3, 199–207.

[24] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens, *CakeML: A verified implementation of ML*, Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (New York, NY, USA), POPL '14, ACM, 2014, pp. 179–191.

[25] Peter J Landin, *The mechanical evaluation of expressions*, The Computer Journal **6** (1964), no. 4, 308–320.

[26] J. Launchbury, *A natural semantics for lazy evaluation*, Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1993, pp. 144–154.

[27] Xavier Leroy, *The ZINC experiment: An economical implementation of the ML language*, (1990).

[28] ______, *The CompCert C verified compiler*, Documentation and user's manual. INRIA Paris-Rocquencourt (2012).

[29] Simon Marlow and Simon Peyton Jones, *Making a fast curry: push/enter vs. eval/apply for higher-order languages*, Journal of Functional Programming **16** (2006), no. 4-5, 415–449.

[30] Alan Mycroft, *Abstract interpretation and optimising transformations for applicative programs*, Ph.D. thesis, 1982.

[31] Keiko Nakata and Masahito Hasegawa, *Small-step and big-step semantics for call-by-need*, Journal of Functional Programming **19** (2009), no. 6, 699–722.

[32] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan, *Functional big-step semantics*, Programming Languages and Systems (Berlin, Heidelberg) (Peter Thiemann, ed.), Springer Berlin Heidelberg, 2016, pp. 589–615.

[33] Simon Peyton Jones, Will Partain, and André Santos, *Let-floating: Moving bindings to give faster programs*, Proceedings of the First ACM SIGPLAN International Conference on Functional Programming (New York, NY, USA), ICFP '96, ACM, 1996, pp. 1–12.

[34] Simon L. Peyton Jones, *Implementing lazy functional languages on stock hardware: The spineless tagless G-machine*, Journal of functional programming **2** (1992), no. 2, 127–202.

[35] Simon L Peyton Jones and David R Lester, *Implementing functional languages*, Prentice-Hall, Inc., 1992.

[36] Guillermo Juan Rozas, *Taming the Y operator*, ACM SIGPLAN Lisp Pointers (1992), no. 1, 226–234.

[37] Amr Sabry, Andrew Lumsdaine, and Ronald Garcia, *Lazy evaluation and delimited control*, Logical Methods in Computer Science **6** (2010).

[38] P. Sestoft, *Deriving a lazy abstract machine*, Journal of Functional Programming **7** (1997), no. 3, 231–264.

[39] Zhong Shao and Andrew W Appel, *Space-efficient closure representations*, Proceedings of the 1994 ACM Conference on Lisp and Functioanl Programming, ACM, 1994.

[40] Konrad Slind and Michael Norrish, *A brief overview of HOL4*, International Conference on Theorem Proving in Higher Order Logics, Springer, 2008, pp. 28–32.

[41] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich, *Total Haskell is reasonable Coq*, Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (New York, NY, USA), CPP 2018, ACM, 2018, pp. 14–27.

[42] George Stelle and Darko Stefanovic, *Verifiably lazy: Verified compilation of call-by-need*, Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (New York, NY, USA), IFL 2018, ACM, 2018, pp. 49–58.

[43] George Stelle, Darko Stefanovic, Stephen L. Olivier, and Stephanie Forrest, *Cactus environment machine*, Trends in Functional Programming (Cham) (David Van Horn and John Hughes, eds.), Springer International Publishing, 2019, pp. 24–43.

[44] David A Terei and Manuel MT Chakravarty, *An LLVM backend for GHC*, ACM Sigplan Notices, vol. 45, ACM, 2010, pp. 109–120.

[45] DA Turner, *Total functional programming*, Journal of Universal Computer Science **10** (2004), no. 7, 751–768.

[46] Philip Wadler, *Monads for functional programming*, Advanced Functional Programming (Berlin, Heidelberg) (Johan Jeuring and Erik Meijer, eds.), Springer Berlin Heidelberg, 1995, pp. 24–52.

[47] Philip Wadler and R John M Hughes, *Projections for strictness analysis*, Functional Programming Languages and Computer Architecture, Springer, 1987, pp. 385–407.

[48] Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg, *A specification for dependent types in Haskell*, Proc. ACM Program. Lang. **1** (2017), no. ICFP, 31:1–31:29.

[49] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr, *Finding and understanding bugs in C compilers*, Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (New York, NY, USA), PLDI '11, ACM, 2011, pp. 283–294.